Vivado Design Suite User Guide

High-Level Synthesis

UG902 (v2012.2) July 25, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
7/25/12	1.0	Initial Xilinx release of the Vivado Design Suite User Guide: High-Level Synthesis.

XILINX®

Table of Contents

Chapter 4: High-Level Synthesis Coding Style Guide

Chapter 5: High-Level Synthesis Command Reference Guide

Appendix A: Additional Resources

High-Level Synthesis Introduction

High-Level Synthesis Introduction

This guide explains different concepts associated with High-Level Synthesis (HLS) and gives a basic overview of High-Level Synthesis and the Xilinx® High-Level Synthesis tool.

- High-Level Synthesis transforms a C, C++ or SystemC design specification into a Register Transfer Level (RTL) implementation which in turn can be synthesized into a Xilinx Field Programmable Gate Array (FPGA).
- Coding style that explains how you can write C code (including C++ and SystemC) for implementation on a Xilinx® FPGA device.
- High-Level Synthesis Reference information.

High-Level Synthesis User Guide

Introduction

The introduction explains different concepts associated with High-Level Synthesis (HLS) and gives a basic overview of High-Level Synthesis, the Xilinx® HLS tool.

Functional Abstraction Level

Definitions

The FPGA design community has moved through a few abstraction levels, to manage the complexity of the designs. Each new abstraction level hides some of the complexity of a design implementation step, offering productivity at the cost of less visibility in the challenges associated with the lower abstraction level:

- A transistor layout database hides the challenges in mask making and wafer processing. The focus of the layout abstraction layer is to respect Design Rule Checks (DRC) which models the basic layout.
- For FPGA design, a netlist avoids a detailed layout effort: the netlist is constructed with instances from a pre-built library. The focus of the netlist abstraction layer is to define the Boolean functionality of the design with appropriate area, performance and power.
- A Register Transfer Level (RTL) description captures the desired functionality by defining datapath and logic between boundaries of registers. RTL synthesis creates a netlist of Boolean functions to implement the design. The focus of the RTL abstraction layer is to define a model for the hardware which is functionally correct.
- A functional specification removes the need to the define register boundaries (and the specific logic required between them) to implement the desired algorithm. The focus of the designer is only on specifying the desired functionality.

As with previous moves up the abstraction level, using a functional specification with high-level synthesis (HLS) to automatically create the RTL design provides productivity benefits in both verification and design optimization.

- The significant benefits of acceleration in simulation time by using a functional C language based specification and the resultant earlier detection of design errors has been embraced for quite a while.
- High Level Synthesis shortens the previous manual RTL creation process and avoids translation errors by automating the creation of the RTL from the functional specification.
- High Level Synthesis automates the optimization of the RTL architecture, allowing multiple architectures to quickly and easily be evaluated before committing to an optimum solution.

C-based Specification

C-based entry is the most popular mechanism to create functional specifications. Currently, ANSI-C (with C99), C++ and SystemC are standards deployed by many system architects to define the functionality of systems intended to be implemented on an FPGA.

High-Level Synthesis provides comprehensive support for C, C++ and SystemC, the IEEE standard (IEEE-1666) used for modeling and concurrent simulation of hardware. The constructs which cannot be synthesized are those which unbounded at elaboration time and for which a finite sized description cannot be determined.

Native C data types live within the classic boundaries of 8-bit, 16-bit, 32-bit and 64-bit words (char, short, int, long, long long). Neither ANSI-C nor C++ has built-in data types to deal with bit-accurate calculations, where the exact bit-width of the data type is used (and which results in optimally sized hardware). High-Level Synthesis provides support for arbitrary precision data types in both C and C++. High-Level Synthesis fully supports the arbitrary precision data types provided by SystemC.

High-Level Synthesis (HLS)

The synthesis of C into RTL employs many advanced transformations working on all aspects of the design area and performance. High-Level Synthesis provides synthesizable support for a large subset of all three input C standards (C, C++ and SystemC) enabling it to synthesize the C code with minimal modifications.

High-Level Synthesis performs two distinct types of synthesis upon the design:

- Algorithm Synthesis takes the content of the functions, and synthesizes the functional statements into RTL statements over a number of clock cycles.
- Interface Synthesis transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system.
    - Interface synthesis can be performed on global variables, top-level function arguments and the return value of the top-level function.

- The types of available interfaces are:
  - Wire
  - Register
  - One-way & two-way handshakes
  - Bus
  - FIFO
  - RAM
- In addition, a function level protocol can be synthesized to the top-level function. The function level protocol includes signals which control when the function can start operation and indicate when it has completed.

High-Level Synthesis synthesis is executed in multiple steps. The effect of interface synthesis impacts what is achievable in algorithm synthesis and vice versa. Like the numerous decisions made during any manual RTL design, the number of available implementations and optimizations is large and the combinations of how they impact each other is very large. High-Level Synthesis abstracts the user away from these details and allows the user to productively get to the best design in the shortest time.

To better understand how High-Level Synthesis is able to abstract the designer away from the implementation details, it is recommended to review the remainder of this section which explains some of the fundamental concepts of HLS and type of optimizations High-Level Synthesis provides:

- Control and Datapath Extraction
- Scheduling & Binding
- Arbitrary Precision Data Types
- Optimizations
- Design Constraints

Control and Datapath Extraction

The first thing which is performed during HLS is to extract the control and datapath inferred by the code. Figure 2-1 shows a small example on how this is performed.

The control functionality is provided by the loops and conditional branches in the code. Figure 2-1 shows how the control behavior can be extracted from the code. Each time the function requires an entry or exit from a loop, it is equivalent to entering or exiting a state in an RTL Finite State Machine (FSM)2.

In Figure 2-1 it is assumed that all operations take a single cycle (or state) to complete. In reality, timing delays and the clock frequency may require more cycles to complete the operations, for example state 1 may expand to states 11, 12 and 13, the control logic may

be impacted by the IO protocols inferred by interface synthesis and High-Level Synthesis may create a more complex and optimized state machine.

The datapath extraction is more straightforward and can be determined by unrolling all the loops and evaluating the conditional statements in the design.

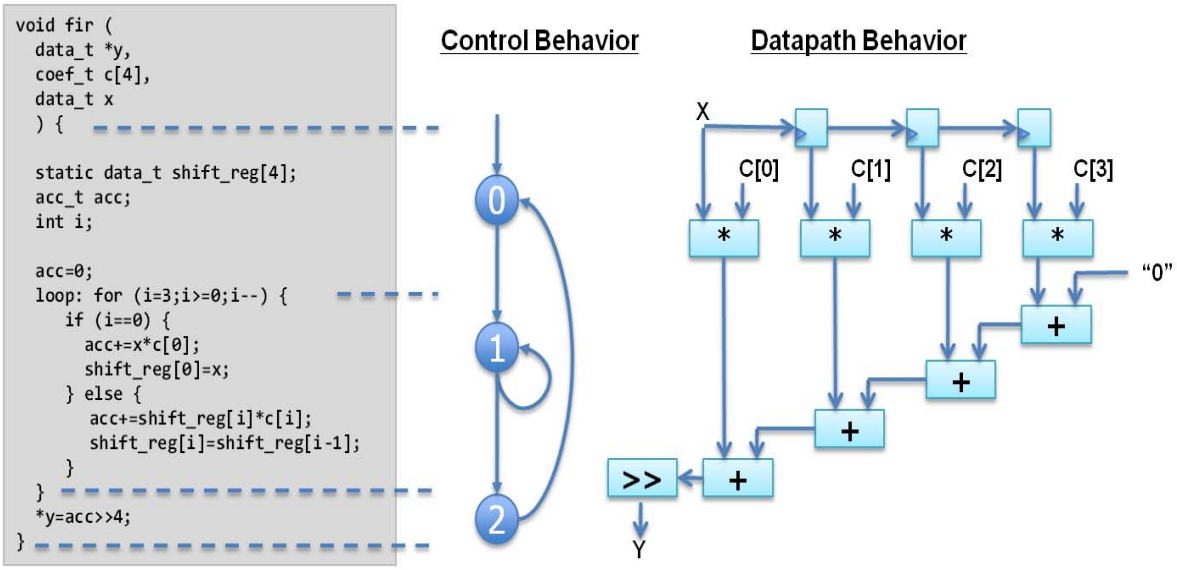

Figure 2-1: Control and Data Extraction

The final datapath implementation in the RTL is unlikely to be as simple as that shown in Figure 2-1: High-Level Synthesis will easily determine that the first adder is not required since the final shift operation is a power of 2 and requires no hardware. More complex optimizations and decisions will be made when the design is scheduled.

Scheduling & Binding

Scheduling and binding are the processes at the heart of high-level synthesis. High-Level Synthesis will determine during the scheduling process in which cycle operations will occur. The decisions made during scheduling take into account, among other things, the clock frequency and clock uncertainty, timing information from the device technology library, as well as area, latency and throughput directives.

For the same example code shown in Figure 2-1, multiple RTL implementations are possible. Figure 2-2 shows just 3 possible implementations.

1. Using 4 clock cycles means a single adder and multiplier can be used, as High-Level Synthesis can share the adder and multiplier across clock cycles: 1 adder, 1 multiplier and 4 clock cycles to complete.

2. If analysis of the target technology timing indicates the adder chain can complete in 1 clock cycle, a design which uses 3 adders and 4 multipliers but which finish in 1 clock cycle can be realized (faster but larger than option 1).
3. Take 2 clock cycles to finish but use only 2 adders and 2 multipliers (smaller than option 2 but faster than option 1).

High-Level Synthesis quickly creates the most optimum implementation based on its own default behavior and the constraints and directives specified by the user. Later chapters explain how to set constraints and directives to quickly arrive at the most ideal solution for the specific requirements.

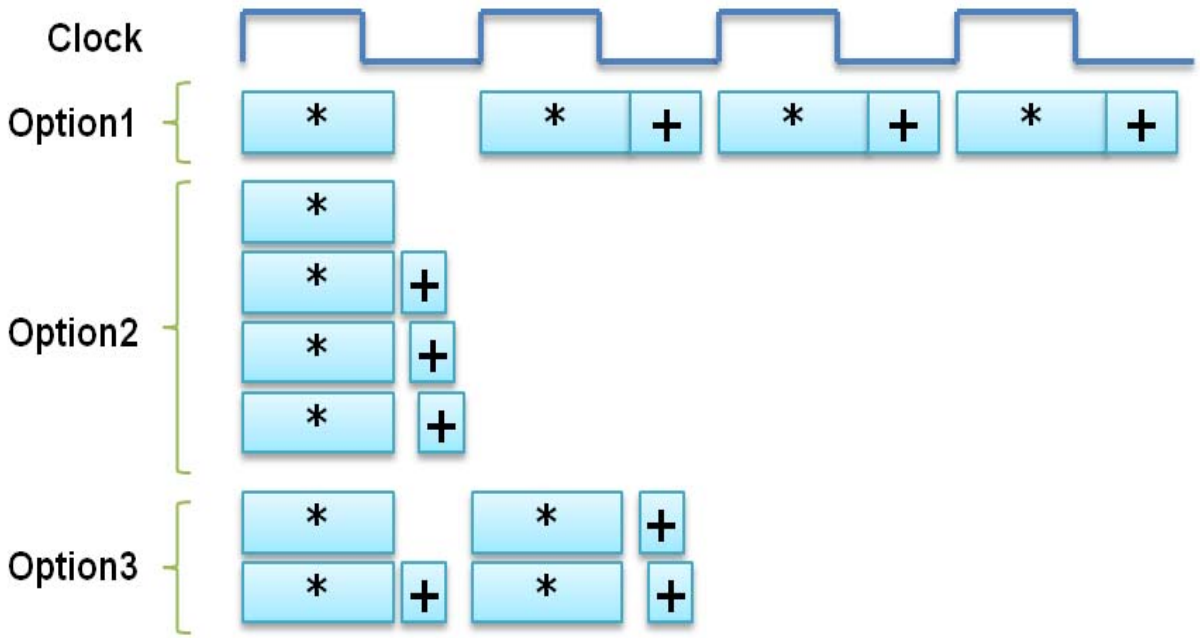

Figure 2-2: Scheduling

Binding is the process that determines which hardware resource, or core, is used for each schedule operation. For example, High-Level Synthesis will automatically determine if an adder and subtractor will used or if a single adder-subtractor can be used for both operations.

Since the decisions in the binding process can influence the scheduling of operations, for example, using a pipelined multiplier instead of a standard combinational multiplier, binding decisions are considered during scheduling.

Arbitrary Precision Data Types

Native C data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL operations (corresponding to hardware) support arbitrary widths. HLS needs a mechanism to allow the

specification of arbitrary precision bit-widths or the RTL design may use 32-bit multipliers when only 17-bit multipliers are required (not an issue to a C program, but a major issue in an RTL design).

High-Level Synthesis provides arbitrary precision integer and fixed-point data types (Table 2-1).

Table 2-1: Integer Data Types

Language	Integer Data Type	Required Header
C	[u]int<precision> (1024 bits)	#include "ap_cint.h"
C++	ap_[u]int<W> (1024 bits) ap_[u]fixed<W,I,Q,O,N>	#include "ap_int.h" #include "ap_fixed.h"
System C	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits) sc_[u]fixed<W,I,Q,O,N>	#include "systemc.h" #include "sc_fixed.h"

These arbitrary types are supported by functions which provide hardware like operations, such as bit-slicing, concatenation and range-selection. Refer to the section “Arbitrary Precision Data Types" section in this User Guide.

Optimizations

High-Level Synthesis can perform a number of optimizations on the design to produce high quality RTL satisfying the performance and area goals. This section introduces a few of the optimization techniques to give an overview of the capabilities.

Pipelining is an optimization which allows one of the major performance advantages of hardware over software, concurrent or parallel operation, to be automatically implemented in the RTL design.

A C program operates in a sequential manner. Given the function "top" shown on the left-hand side of Figure 2-3, every sub-function from “func_A” to “func_C” must complete its operation before “func_A” can once again execute.

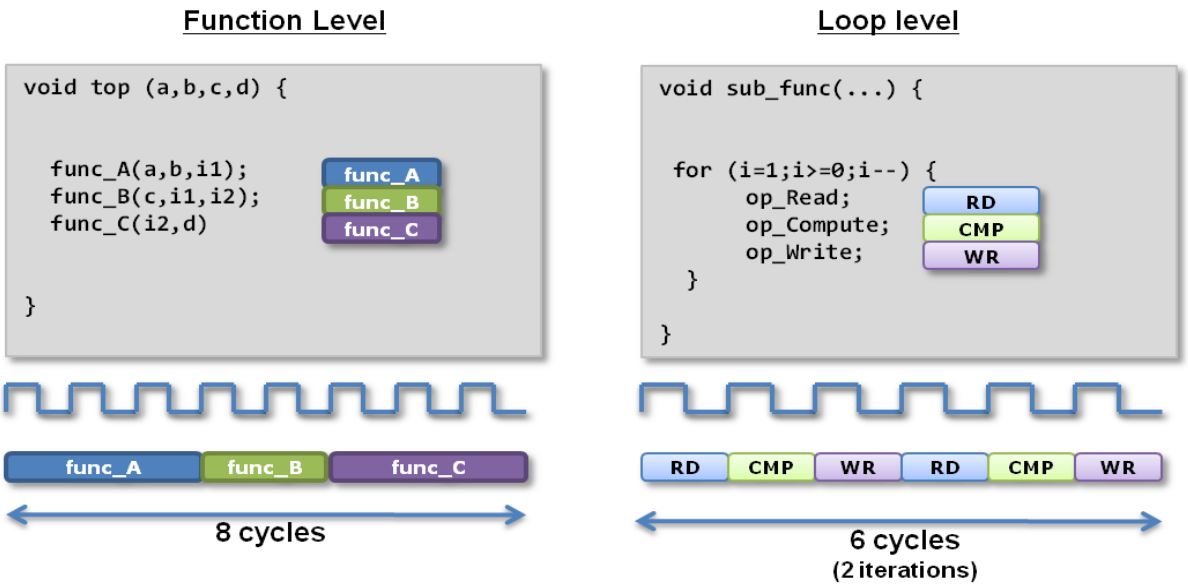

Figure 2-3: Functions & Loops Without Pipelining

Even if "func_A" is ready to process the next set of operations as soon as it is finished, functions "func_B" and "func_C" must complete execution before "func_A" can once again begin operation.

As function "sub_func" on the right-hand side of Figure 2-3 shows, it is the same at the operator level: the first operation cannot re-execute until the last is complete.

The sequential nature of the C language, or in other words its lack of concurrency, puts artificial dependencies on operations which must wait their turn for execution. High-Level Synthesis provides the ability to automatically pipeline both functions and loops to ensure the RTL design does not suffer from such limitations.

By default, High-Level Synthesis will seek to execute these operations in parallel and reduce the overall latency of the design. In addition to this, High-Level Synthesis can improve the throughput by pipelining these operations, allowing different executions of the function or different loop iterations to overlap in time.

Figure 2-4 shows the result when High-Level Synthesis is used to pipeline the sub-functions and/or operations in a loop.

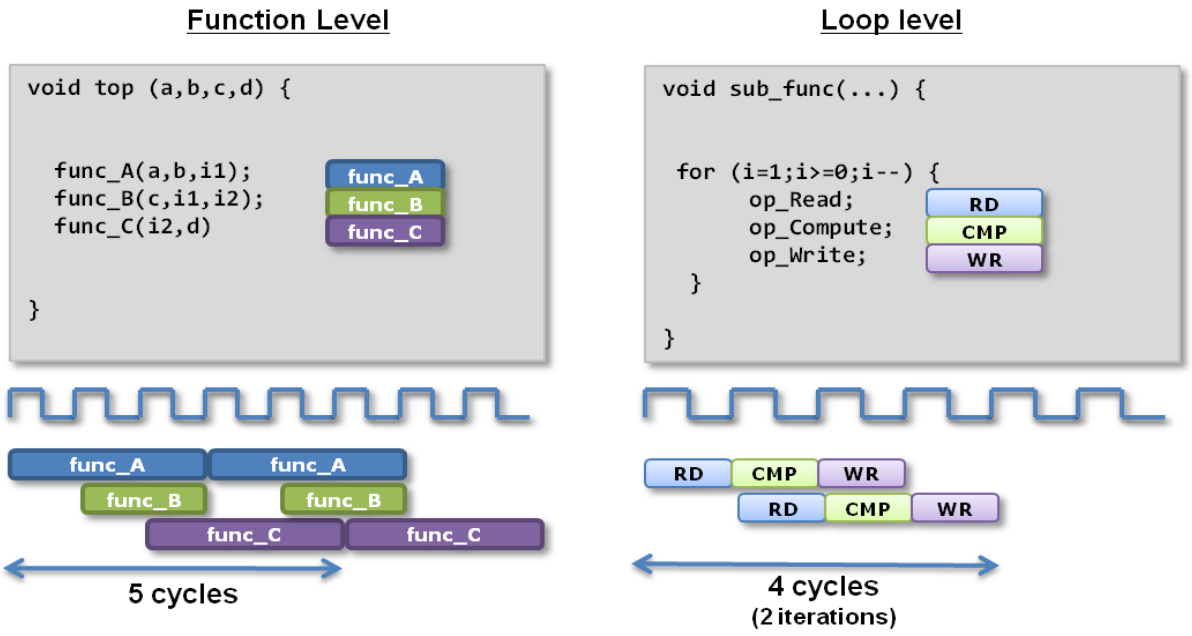

Figure 2-4: Functions & Loops With Pipelining

- At the function level, dataflow optimization allows the sub-functions ("func_A", "func_B" and "func_C") to execute as soon as data is available.
  - Function "func_A" starts it's next operation "before func_C" has completed its first execution.
  - Compared with the previous implementation in Figure 2-3, the 8 clock cycles it took to execute the function is now only 5 cycles and "func_A" starts a new operation every 3 clock cycles instead of every 8.
- Pipelining the loop allows the operations in a loop to execute concurrently.
  - Figure 2-4 shows how loop pipelining can also positively performance compared with Figure 2-3: the loop completes in only 4 clock cycles and processes an new input (RD operation) every clock cycle instead of waiting for 3 clock cycles.

Another example of a design optimization which can be automatically implemented by High-Level Synthesis is array partitioning.

Within C language descriptions, arrays are used as a convenient way to group similar elements together. When the elements of arrays are synthesized as storage elements (that is, when the value must be maintained across clock cycles) these array elements can be grouped at the RTL in RAMs or they can be broken into their constituent parts and implemented as individual registers.

- If the elements of an array are accessed one at a time, an efficient implementation in hardware is to keep them grouped together and mapped into a RAM.

- If multiple elements of an array are required simultaneously, it may be more advantageous for performance to implement them as individual registers: allowing parallel access to the data.

Implementing an array of storage elements as individual registers may help performance but this loses the substantial benefits of RAMs: area efficient in all technologies and they are readily available in the device as BRAMs (separate from the LUTs and registers).

High-Level Synthesis provides a variety of techniques to ensure arrays are implemented in the most optimal manner:

- Partitioning large arrays into multiple smaller arrays, which can be mapped to different instances of RAM (allowing multiple reads or writes in the same cycle).
- Enabling multiple small arrays to be implemented onto the same RAM resource.

The application of a few simple directives provides for a large number of different implementations, from pipelining to the manipulation of arrays, ensuring that the most optimal implementation for the particular design can be quickly and easily found.

Design Constraints

Finally, in addition to the clock period and clock uncertainty, High-Level Synthesis offers a number of constraints including the ability to:

- Specify a specific latency across functions, loops and regions.
- Specify a limit on the number of resources used.
- Override the inherent or implied dependencies in the code and permit operations (for example, a memory read before write)

These constrains can be applied using High-Level Synthesis directives to create a design with the desired attributes.

Designing with High-Level Synthesis is a HLS flow allows the designer to quickly implement an initial architecture, which will be defined by the dependencies in the code and the default High-Level Synthesis interpretation of C language constructs, and then easily direct the design with directives towards the desired high performance implementation.

Introduction to High-Level Synthesis

High-Level Synthesis Overview

As shown in Figure 2-5, High-Level Synthesis accepts as input, a C-based design description, and directives and constraints, specified using the Graphical User Interface (GUI) or a Tcl batch script. A technology library specifying the timing and area details of all supported Xilinx device is built-in and is not required to be supplied.

High-Level Synthesis outputs RTL design files in Verilog, VHDL and SystemC. In addition verification and implementation scripts, used to automated the RTL verification and RTL synthesis steps are also created.

This section provides an overview of these various inputs and outputs.

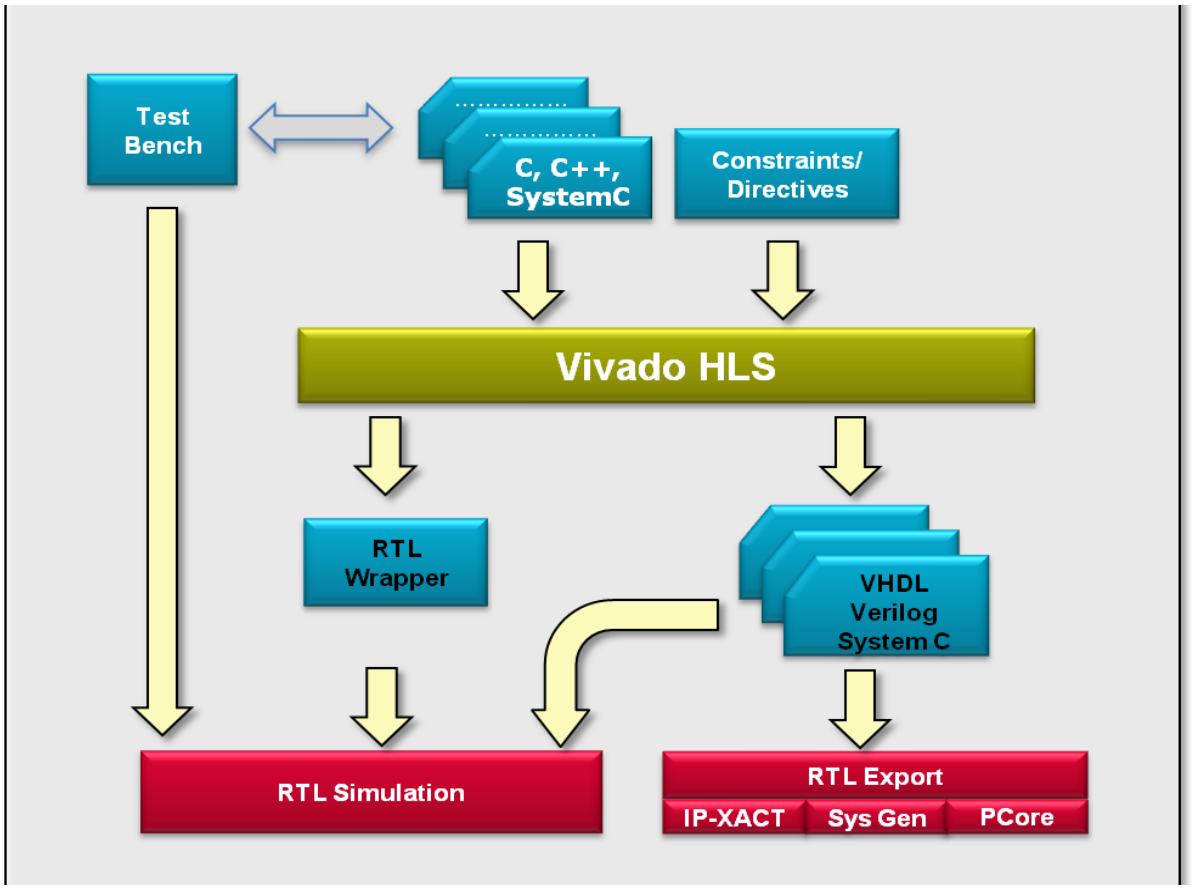

Figure 2-5: High-Level Synthesis Use Model

Design Files

When referring to C, or C-based design, High-Level Synthesis covers all 3 standards:

- ANSI-C enhanced with a data type for arbitrary integer precision.
- C++ enhanced with classes for arbitrary integer precision and fixed point precision.
- SystemC (IEEE-1666)

The documentation will elaborate on how to simulate the input specification, including explanations of the provided arbitrary precision enhancements.

The C-based input can include a test bench. If provided, a C test bench can be re-used to verify the output RTL: improving designer productivity by removing the need to create RTL test benches for RTL verification. High-Level Synthesis supports multiple input files and while the recommended flow separates the test bench from the design to be synthesized in separate files, this is not required.

Device Technology Library

A device technology library models the area and timing of each supported Xilinx device, enabling the optimization engine to make the appropriate trade-offs. The device technology library is built-in to High-Level Synthesis and does not need to be supplied.

Directives and Constraints

The directives and constraints are specified in the High-Level Synthesis GUI or with the Tcl-based command language and drive the optimization engine towards the desired performance goals and RTL architecture.

RTL Output

The RTL output is written automatically after the successful completion of synthesis. High-Level Synthesis supports three hardware description language standards:

- VHDL(IEEE 1076-2000)
- Verilog(IEEE 1364-2001)
- SystemC(IEEE 1666-2006 -Version 2.2-)

Note: The SystemC output from High-Level Synthesis is the design implementation at the Register Transfer Level (RTL).

Simulation Output (RTL co-simulation)

High-Level Synthesis creates the scripts required to verify the generated RTL through co-simulation with the original test bench and a variety of RTL simulators. The following RTL simulators are supported:

- ModelSim
- VCS
- OSCI SystemC

The SystemC output can be verified using the built-in SystemC kernel and requires not third part simulator or license. The supported HDL simulators require a license from the appropriate vendor.

Implementation Output

The scripts and constraint files required for processing the design through RTL synthesis and P&R on the FPGA are provided. These scripts ensure the RTL synthesis process can be completed in a push-button manner from the High-Level Synthesis GUI.

Using High-Level Synthesis

This section provides an introduction to High-Level Synthesis, explaining how to invoke High-Level Synthesis, create a project, use solutions to manage the RTL implementation and apply directives for optimization. After this introduction, details on a tutorial example are provided.

High-Level Synthesis can be invoked as a Graphical User Interface (GUI) or as a Command Line Interface (CLI) which accepts Tcl commands in interactive or batch mode.

High-Level Synthesis Graphical User Interface

Windows

To invoke High-Level Synthesis on a PC Windows platform double-click on the desktop icon as shown in Figure 2-6.

Figure 2-6: **Vivado HLS GUI Icon**

Linux

To invoke High-Level Synthesis on a Linux platform execute the following command at the Linux command prompt.

```
$ vivado_hls
```

The High-Level Synthesis GUI invokes as shown in Figure 2-7.

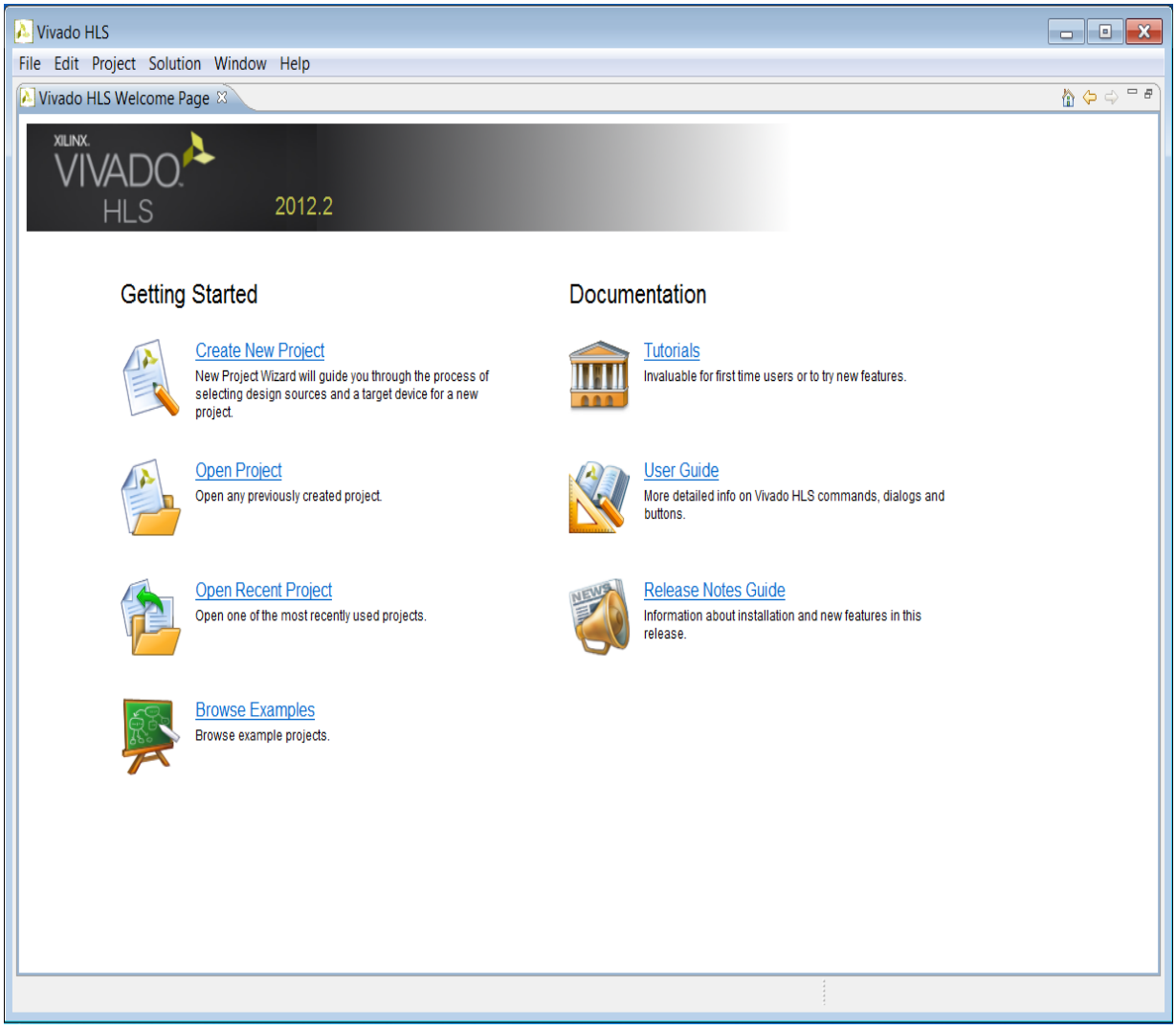

Figure 2-7: **GUI Mode**

The **`Getting Started`** options in Figure 2-7 allow the following tasks to be performed:

- Create New Project
  - This will launch the project setup wizard.

- Open project
  - Navigate to an existing project.
- Open Recent Project
  - Select from a list of recent projects.

The **Documentation** tasks available directly from the **Welcome Screen** (Figure 2-7) are:

- Browse Examples
  - Open High-Level Synthesis examples. These can also be found in the examples directory in the High-Level Synthesis installation area.
- Release Note Guide
  - Open the Release Notes for this version of software.
- User Guide
  - Open the High-Level Synthesis User Guide.
- High-Level Synthesis Tutorial
  - Select a tutorial to open.

High-Level Synthesis Command Line Interface

On Windows the High-Level Synthesis Command Line Interface (CLI) can be invoked from the start menu: `Xilinx Design Tools > vivado 2012.2 > Vivado HLS Command Prompt`.

Figure 2-8: Vivado HLS CLI Icon

On Windows and Linux, using the `-i` option with the `vivado_hls` command will open High-Level Synthesis in interactive mode. High-Level Synthesis will wait for Tcl commands to be entered.

```
$ vivado_hls -i [-l <log_file>]

vivado_hls>
```

By default, High-Level Synthesis creates an `vivado_hls.log` file in the current directory. To specify a different file, the `-l <log_file>` option can be used.

High-Level Synthesis supports auto-completion: press the <TAB> key after the initial few letters of a command and High-Level Synthesis will offer a list of candidates that match the command.

```
vivado_hls> open<TAB>
open
open_project
open_solution
```

The High-Level Synthesis commands have built-in help, which can be accessed with the help command in High-Level Synthesis. The command name can still be provided through auto-complete:

```
vivado_hls> help <command>
```

Type the exit command to quit interactive mode, and return to the shell prompt:

```
vivado_hls> exit
$
```

Commands also can be embedded in a Tcl script and executed in batch mode with the -f <script_file> option.

```
$ vivado_hls -f script.tcl
```

To further help with script automation High-Level Synthesis provides options which will return details on the environment in which it is running, namely the -version option which returns the version number of High-Level Synthesis, -system which returns operating system High-Level Synthesis is running on, the -machine option which returns the current machine architecture and -root_dir which returns the name of the directory where High-Level Synthesis is installed.

Using the CLI Shell on Windows

On the Windows OS, the CLI shell is implemented using the Minimalist GNU for Windows (minGW) environment which allows both standard Windows DOS commands to be used and/or a subset of Linux commands to be used.

Figure 2-9 shows that both (or either) the Linux ls command and the DOS dir command can be used to list the contents of a directory.

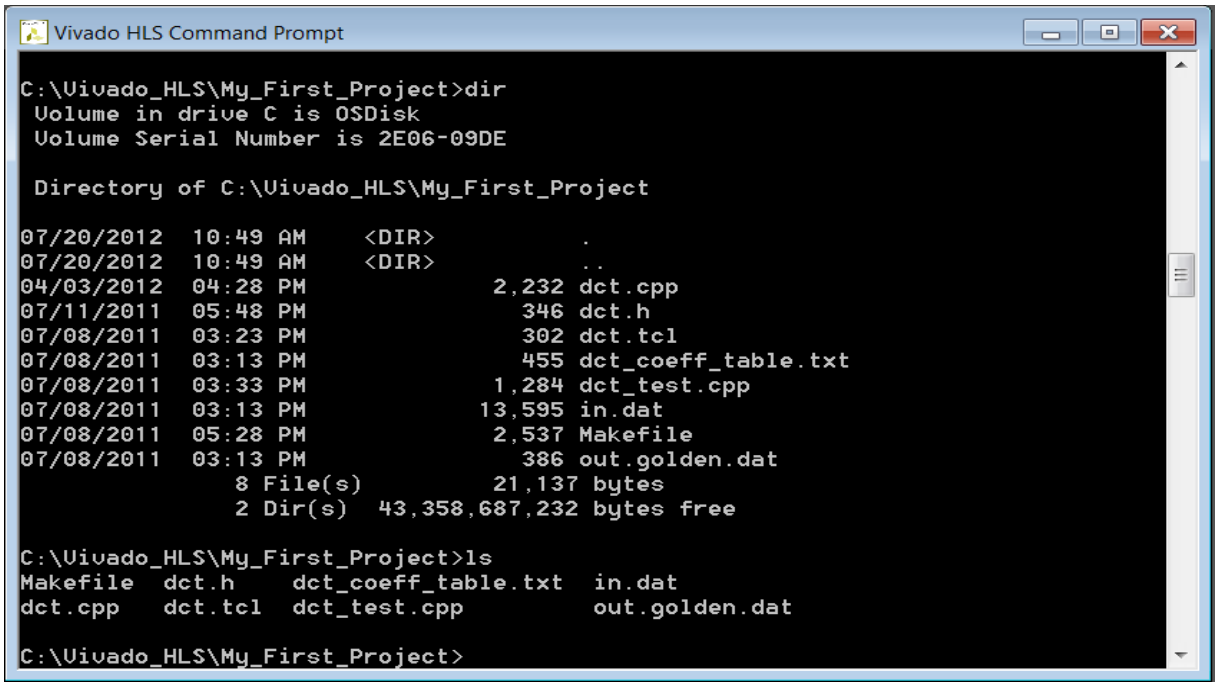

Figure 2-9: High-Level Synthesis CLI Icon

Be aware that not all Linux commands and behaviors are supported in the minGW environment. The following represent some know common differences in support:

- The Linux which command is not supported.
- Linux paths in Makefile will be automatically expanded to minGW paths. In all Makefile, replace any Linux style pathnames assignments such as FOO := :/ with versions where the pathname is quoted such as FOO := ":/" to prevent any path substitutions.

Creating an High-Level Synthesis Project

The first step in using High-Level Synthesis is to create a new project or open an existing project. As shown in Figure 2-7 when the High-Level Synthesis GUI invokes the menu commands for performing these operations are File > New Project and File > Open Project.

When File > New Project is selected the High-Level Synthesis project wizard invokes. The first screen of the project wizard asks for details on the project specification as shown in Figure 2-10.

The fields for entering the project specification are:

- **Project Name**: In addition to being the project name this will be the name of the directory when the project details are stored. The use of a file extension, such as the

.prj extension shown in Figure 2-10, makes the directory easily identifiable as a project directory but is not a requirement.

- **Location**: This is when the project will be stored.
- **Top Level**: If the top-level module is a SystemC SC_MODULE, select SystemC. Otherwise select C/C++ (the default). If the source files use SystemC types but the top-level is not an SC_MODULE, select C/C++.

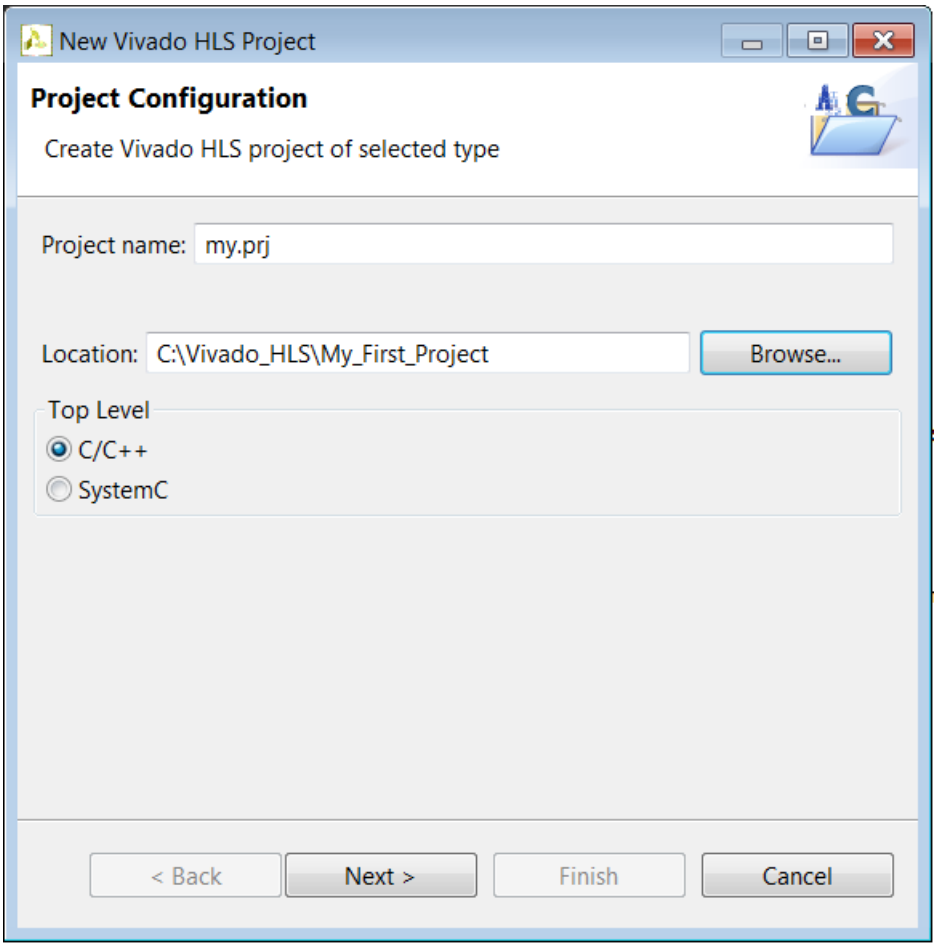

Figure 2-10: Project Specification

Pressing the Next > button will move the wizard to the second screen where details of the project C sources files can be entered (Figure 2-11).

The name of the top-level function to be synthesized should be specified.

Note: This is not required when the project is specified as SystemC.

Use the Add Files... button to add the source code files to the project.

The Edit CFLAGS button allows any C compiler flags required to successfully compile the source files, to be added to the project.

Examples of C compiler flags include macro specifications such as –DMACRO_1 which defines macro MACRO_1 during compilation, -fnested-functions which is required for any design which contains nested functions and –I/project/source/headers which provides the search path for any associated header files. Any headers which exist in the local directory (as specified by the Location in Figure 2-10) are automatically found and included.

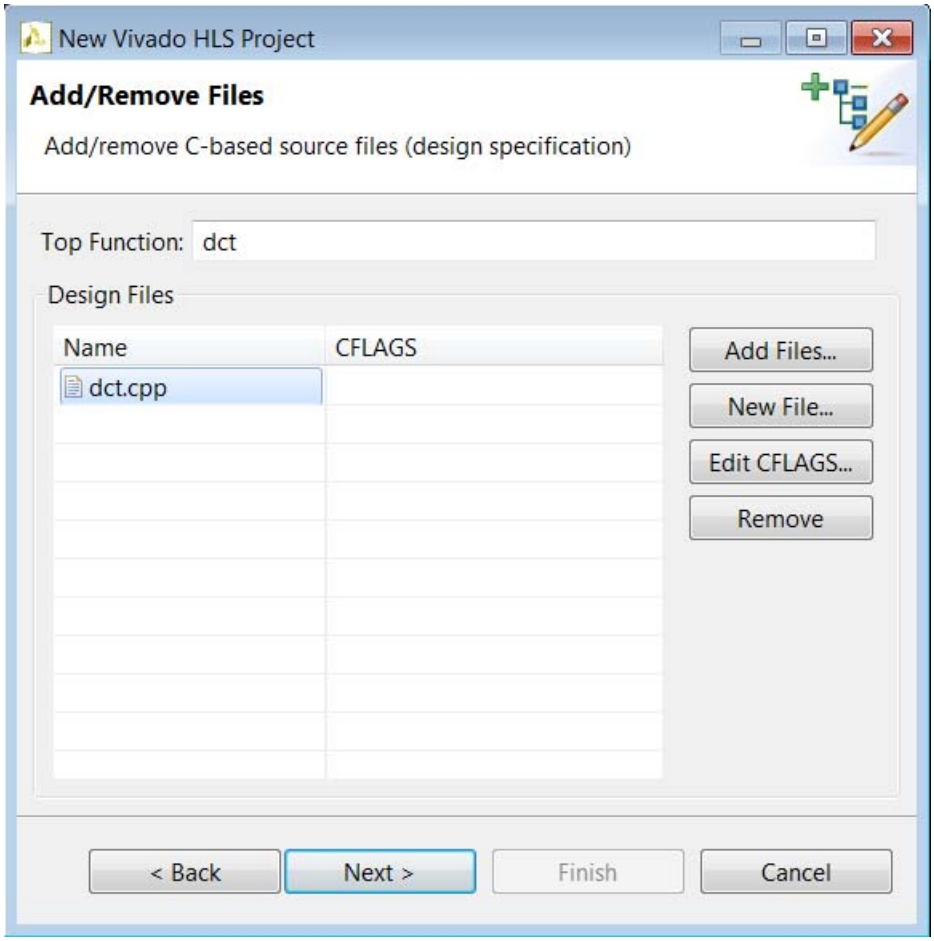

Figure 2-11: Project Source Files

The next window in the project wizard allows the files associated with the test bench to be added to the project.

The C test bench used to validate the C algorithm can be reused to verify the output RTL. High-Level Synthesis automatically creates the adapters and wrappers to instantiate the RTL design into the C test bench and verify the RTL through co-simulation of C and HDL, negating the requirement to create an RTL test bench.

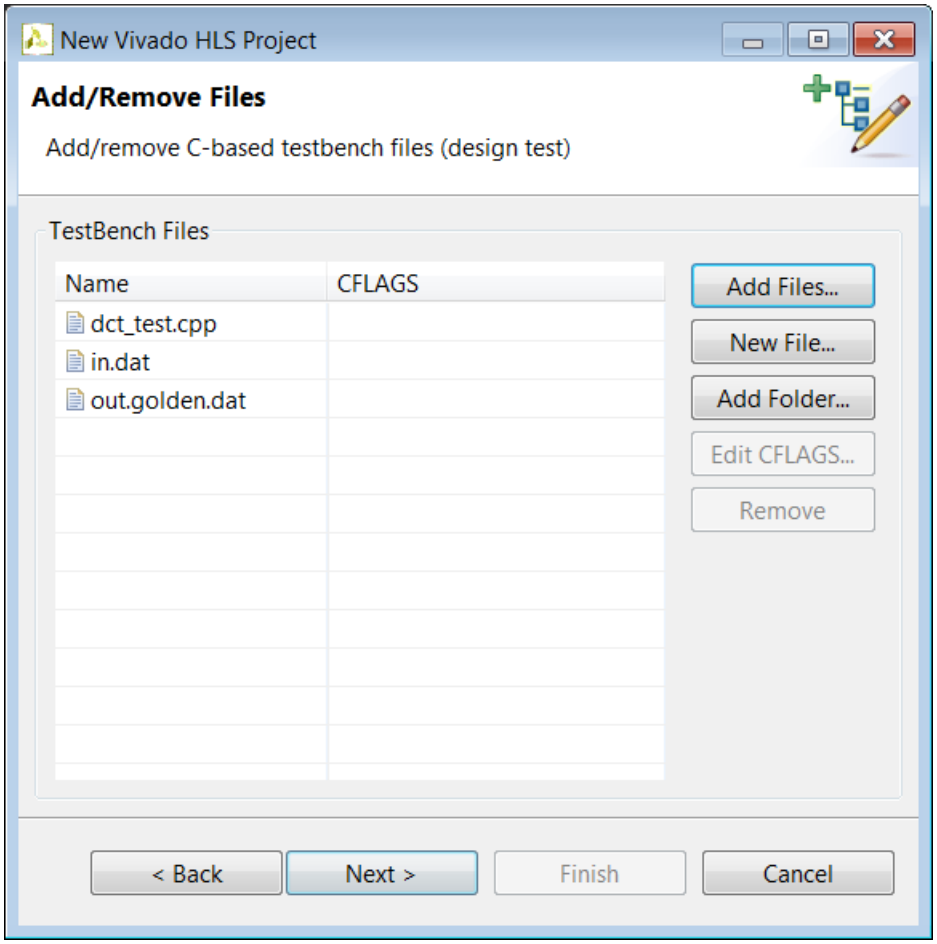

Figure 2-12: Project Test Bench Files

As with the C source files, the Add Files... button is used to add the C test bench and the Edit CFLAGS button to include any C compiler options.

In addition to the C source files, all files read by the test bench should be added to the project. In the example shown in Figure 2-12, the test bench opens file in.dat to supply input stimuli to the design and file out.golden.dat to read the expected results. Since the test bench accesses these files, both are (and must be) included in the project.

If the test bench files exist in a directory, the entire directory may be included rather than the individual files.

If there is no C test bench, there is no requirement to enter any information here and the Next> button will open the final window of the project wizard which allows the details for the first solution to be specified (Figure 2-13)

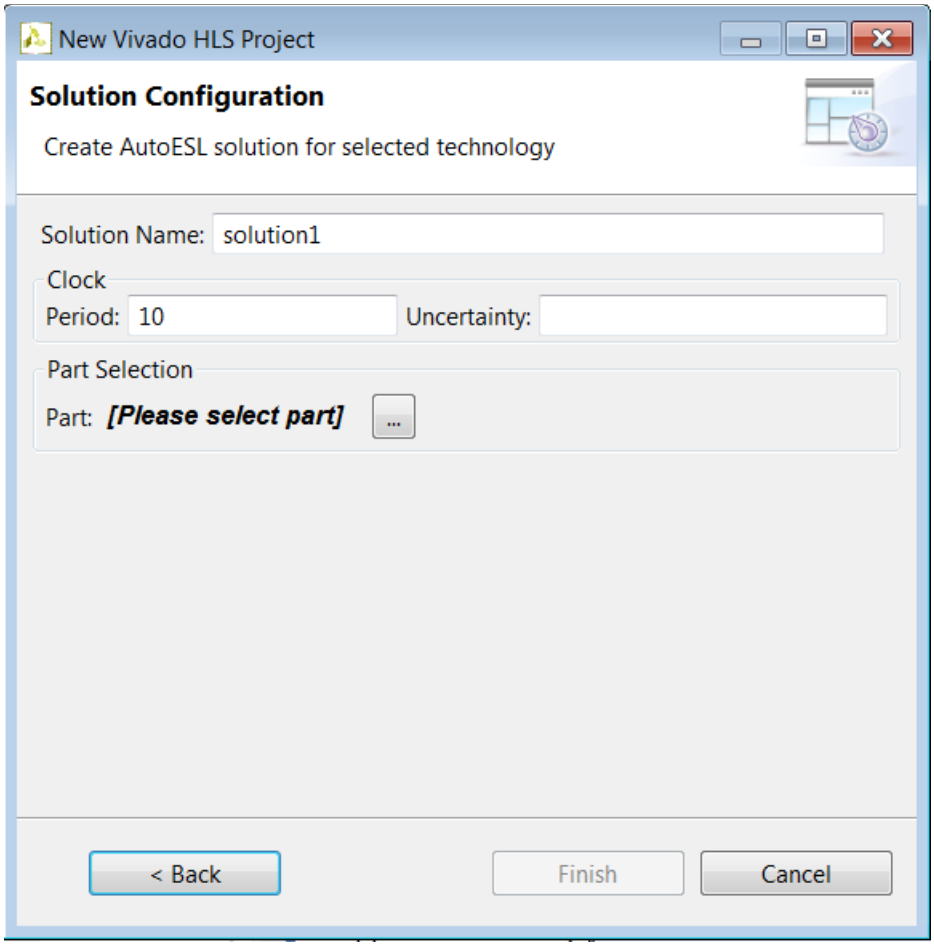

Figure 2-13: Initial Solution Settings

The fields in Figure 2-13 allow the following details to be specified:

- **Solution Name**: High-Level Synthesis provides the initial default name solution1 but any name can be specified for the solution.
- **Clock**: The clock period is specified using units of ns. The clock period used for synthesis is the clock period minus the clock uncertainty. High-Level Synthesis uses the timing information in the technology library to create the RTL design. The clock uncertainty value allows a user controllable margin to account for any increases in net delays due to RTL synthesis, place and route. If not specified in ns, the clock uncertainty defaults to 12.5% of the clock period.
- **Part**: Press to select the appropriate technology (Figure 2-14).

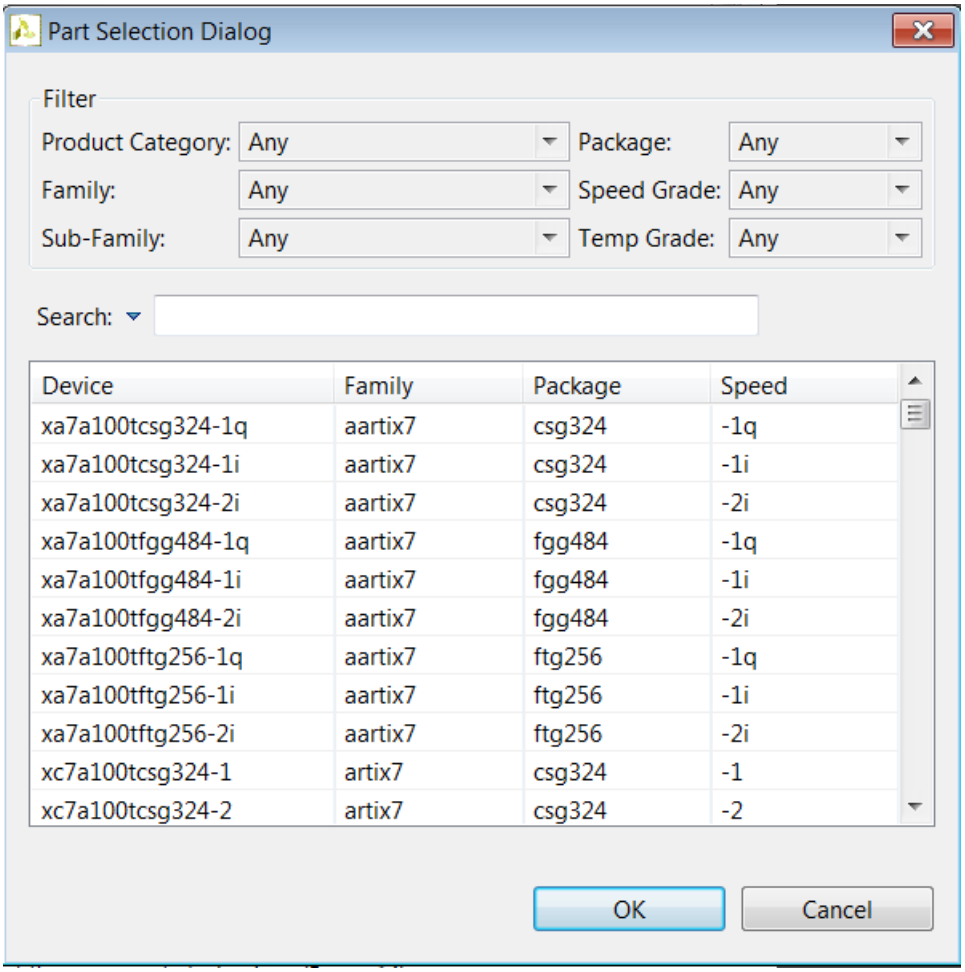

Figure 2-14: Part Selection

Selecting Finish will open the project as shown in Figure 2-15.

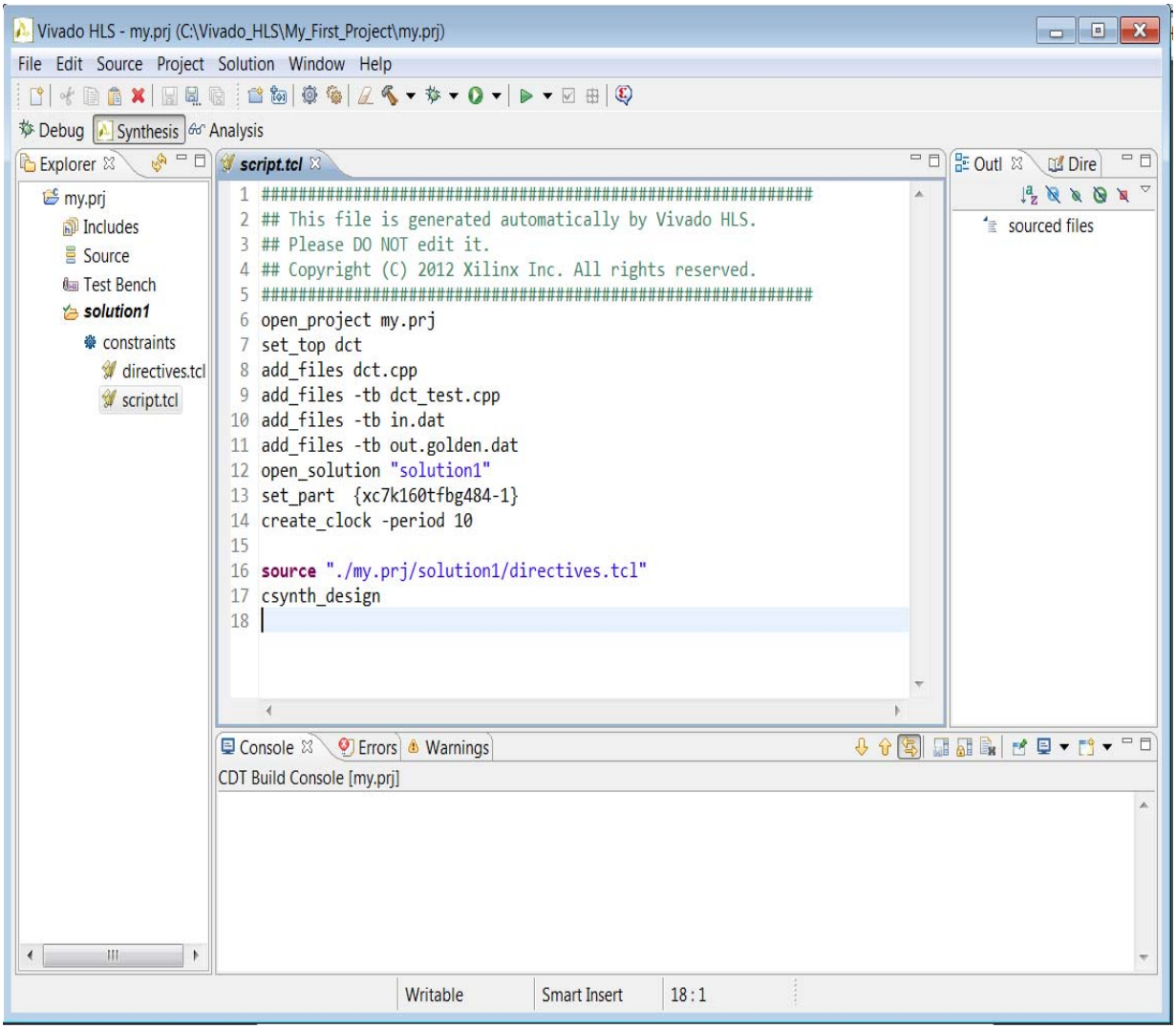

Figure 2-15: New Project

All the Tcl commands for creating the project are stored in the script.tcl file within the solution. Double-clicking on the script.tcl file in the Explorer pane (right-hand side of Figure 2-15) opens the file in the Information pane, as shown in Figure 2-15.

For users wishing to develop Tcl batch scripts, the script.tcl file is an ideal starting point. The containers shown in the Explorer pane can be found in the project directory: simply copy the file from solution directory.

The primary commands for using High-Level Synthesis are provided in the toolbar (Figure 2-16). Project control ensures only commands which can be currently executed are highlighted. For example, synthesis must be performed before RTL simulation can be executed and thus the simulation toolbar button will remain grey until synthesis completes.

The primary command buttons, shown within the red box in Figure 2-16, are (in left to right order):

- Create a New Project
- Create a New Solution
- Edit the existing Project Settings
- Edit the existing Solution Settings
- Compare Solution reports
- Cleanup the C simulation environment.
- Build the C/C++/SystemC executable.
- Run the C/C++/SystemC executable.
- Run the C/C++/SystemC executable in debug mode.
- Run Synthesize
- Run RTL Simulation
- Export RTL Design

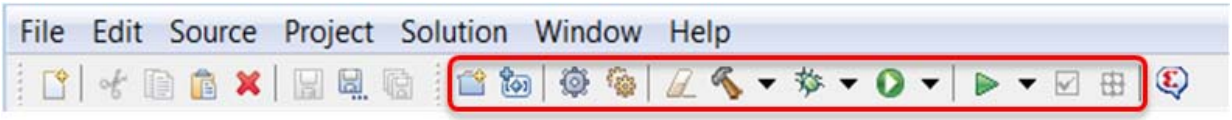

Figure 2-16: Tool Bar

Each of the buttons on the tool bar has an equivalent command in the menus.

The first step after creating a project is to validate the C function. Pressing the Build toolbar button will compile the C design (debug or optimized release version), as shown in Figure 2-17.

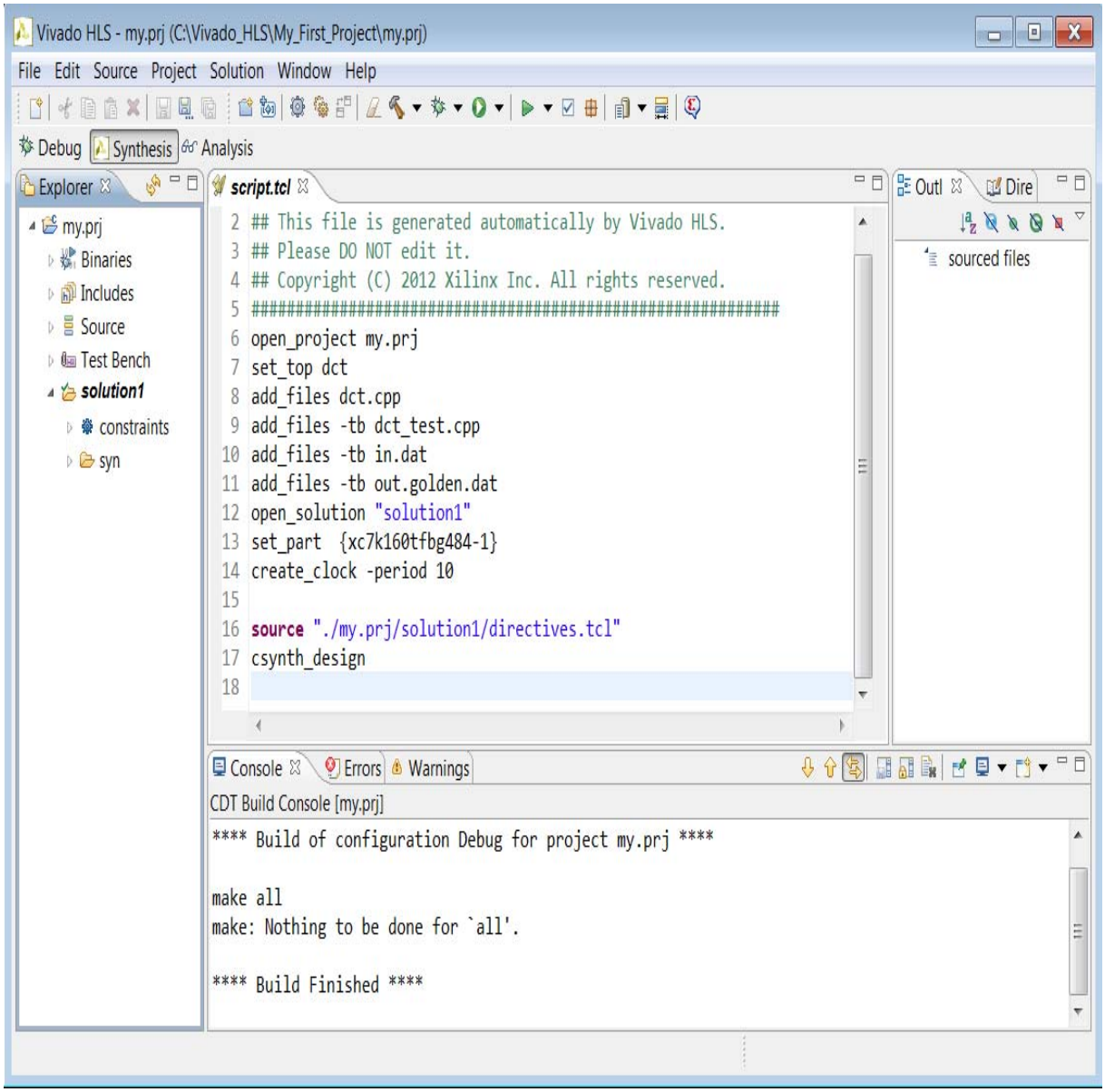

Figure 2-17: C Compiled with Build

The build can then be run or optionally viewed in the debug environment. If the Debug toolbar button is used the debug environment can be opened (Figure 2-18) and the debug step buttons (red box in Figure 2-18) to step through the code and analyze its operation.

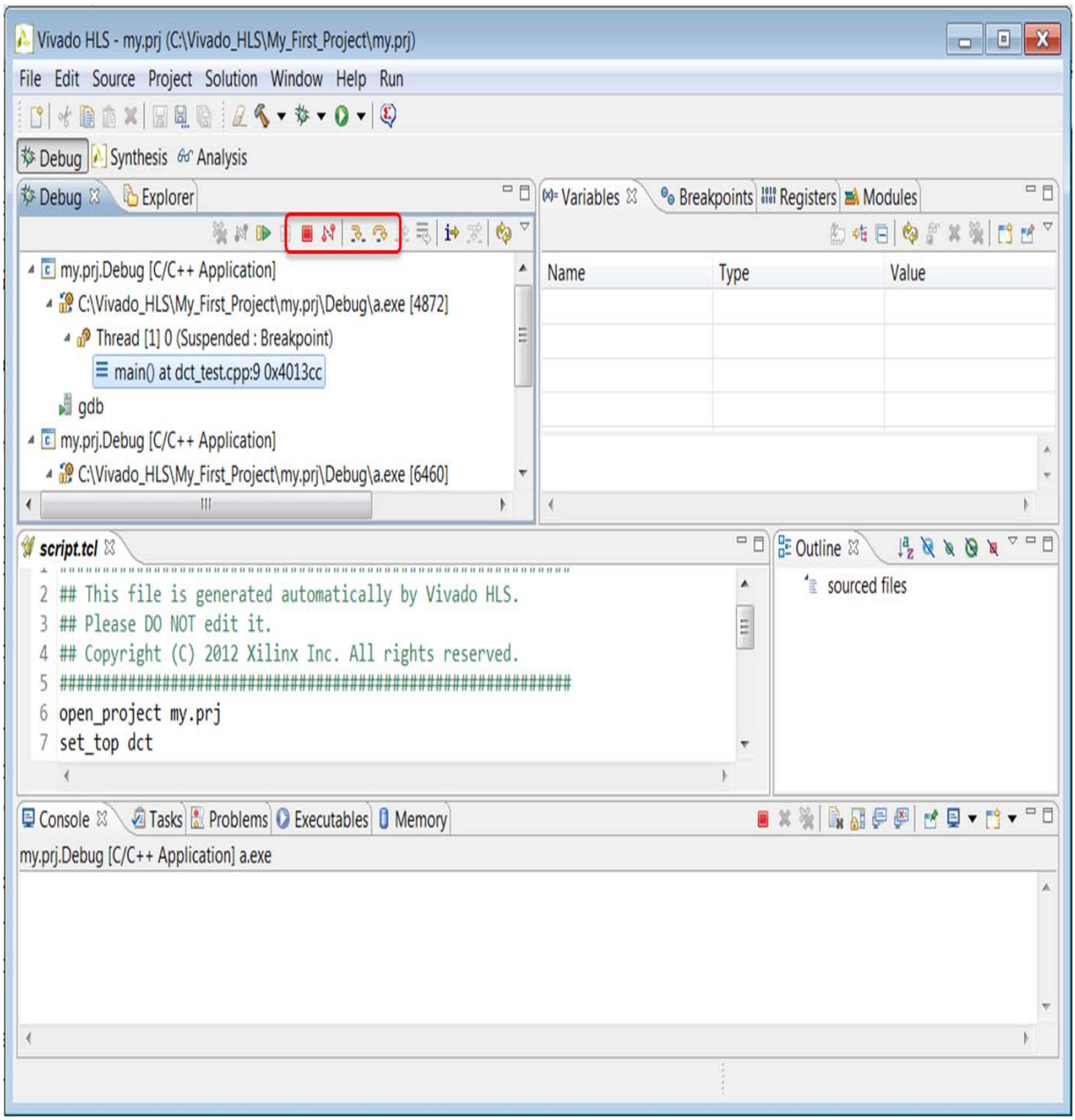

Figure 2-18: C Debug Environments

The next step is to execute synthesis. When synthesis completes the synthesis report is available, it will open automatically in the information pane, and results can be analyzed. (Figure 2-19)

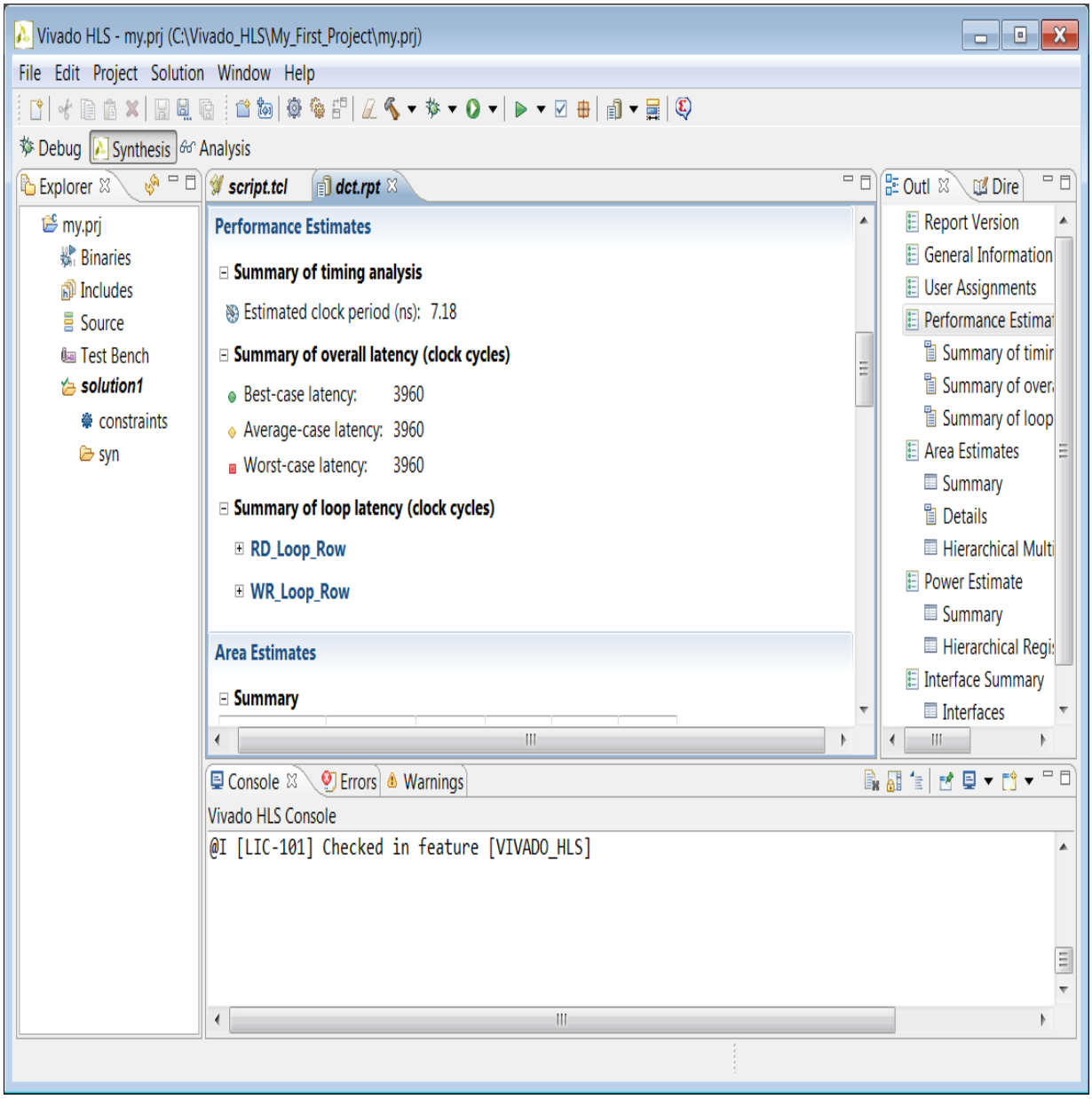

Figure 2-19: Synthesis Report

The report provides details on both the performance and area of the RTL design. The outline tab can be used to navigate through the report. Reports are created for each function in the hierarchy (unless the function was inlined: an optimization discussed in later chapters). The report for the top-level function provides details for the entire design.

Table 2-2 explains the categories in the synthesis report.

Table 2-2: **Synthesis Report Catagories**

Category	Sub-Catagory	Description
Report Version	---	Details on the version of High-Level Synthesis used to create the results.
General Information	---	Project name, solution name and when the solution was executed.
User Assignments	---	Details on the technology, target device attributes and the target clock period.
Performance Estimates	Summary of timing analysis	The estimate of the fastest achievable clock frequency. This is an estimate because logic synthesis and P and R are still to be performed.
	Summary of overall latency	The latency of the design: the number of clock cycles from the start of execution until the final output is written. If the latency of loops can vary, the best, average and worse case latencies will be different. If the design is pipelined this section will show the throughput (Without pipelining the throughput is the same as the latency: the next input will be read when the final output is written).
	Summary of loop latency	This shows the latency of individual loops in the design. The trip count is the number of iterations of the loop. The loop latency is the latency to complete all iterations of the loop.
Area Estimates	Summary	This shows the resources (LUTS, Flip-Flops, DSP48s etc.) used to implement the design. The sub-categories are explained in the Details section of this table.
	Utilization	Shows the utilization of resources for the selected device.

Table 2-2: **Synthesis Report Catagories**

Category	Sub-Catagory	Description
	Details: Component	The resources specified here are used by the components (sub-blocks) within the top-level design. Components are created by sub-functions in the design. Unless inlined, each function becomes it's own level of hierarchy. In this example there are no sub-blocks: the design has one level of hierarchy.
	Details: Expression	This category shows the area used by any expressions such as multipliers, adders, comparators etc. at the current level of hierarchy.
	Details: FIFO	The resources listed here are those used in the implementation of FIFOs at this level of the hierarchy.
	Details: Memory	The resources listed here are those used in the implementation of memories at this level of the hierarchy.
	Details: Multiplexors	All the resources used to implement multiplexors at this level of hierarchy are shown here.
	Details: Registers	This category shows the register resources used at this level of hierarchy.
	Hierarchical Multiplexor Count	A summary of the multiplexors throughput the hierarchy.
Power Estimate	Summary	The expected power used by the device. At this level of abstraction the power is an estimate and should be used for comparing the efficiently of different solutions.
	Hierarchical Register Count	The estimated power used by resisters throughput the design hierarchy.
Interface Summary	Interface	This section shows the details on type of interfaces used for the function and the ports: port names, directions, bit-widths, etc.

The most typical use of High-Level Synthesis is to create an initial design, then perform optimizations to meet the desired area and performance goals. Solutions offer a convenient

way to ensure the results from earlier synthesis rounds can be both preserved and compared.

Using Solutions

The New Solution tool bar button (Figure 2-16) or the menu Project > New Solution can be used to create a new solution. This opens the Solution Wizard (Figure 2-20).

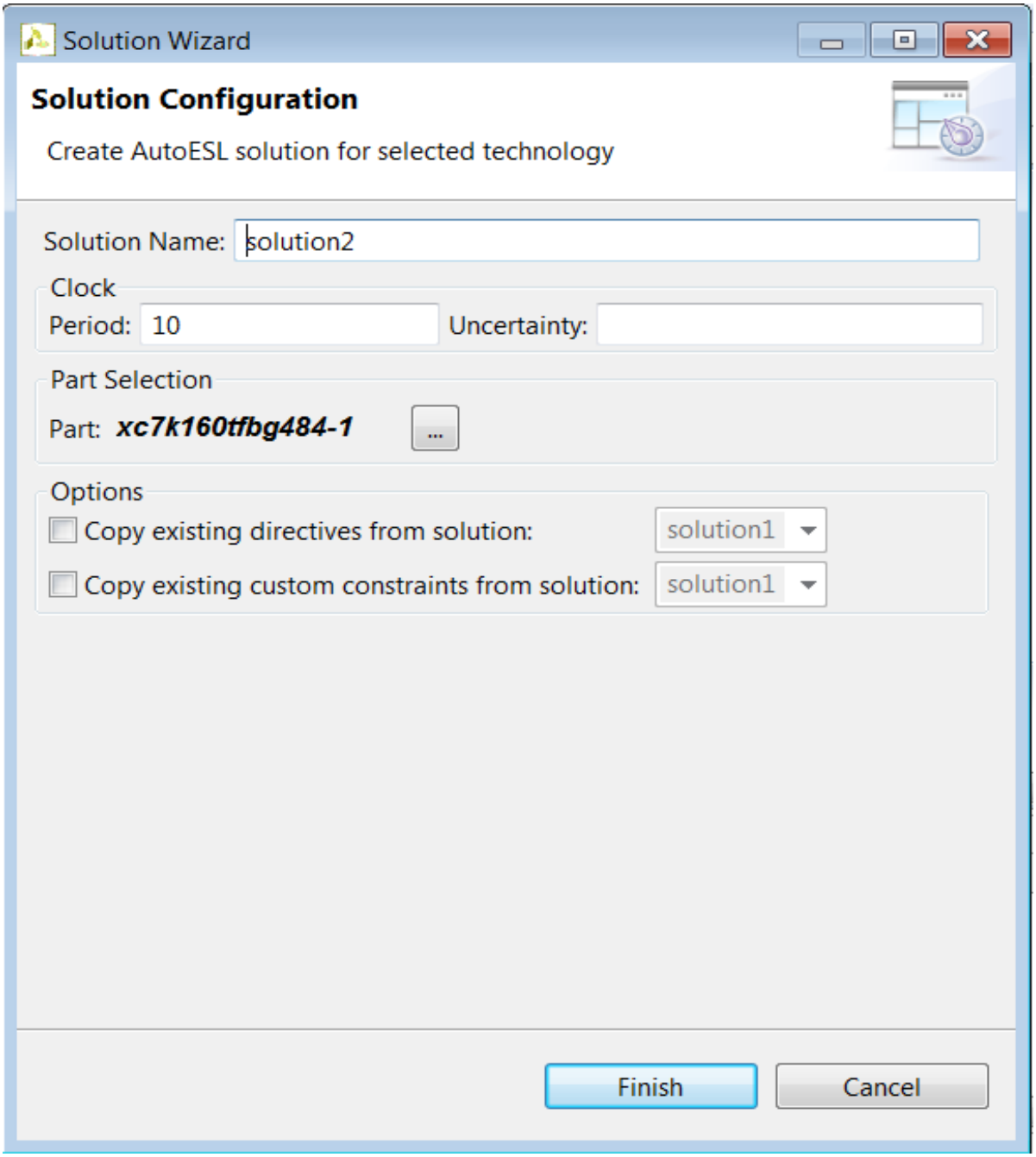

Figure 2-20: Solution Wizard

The solution setting window has the same options as the final window in the New Project wizard (Figure 2-15) plus two additional options which allow directives and customs commands which were applied to a solution to be conveniently copied to the new solution, where they may be modified or removed. The next section explains how directives can be added to solutions.

Using The High-Level Synthesis GUI

Before discussing how optimizations are performed, it is worth spending some time to review how the High-Level Synthesis GUI displays information and how it can be customized.

In some cases the default setting of the High-Level Synthesis GUI may prevent certain information from being shown. This relates to the following:

- Information defined in header files.
- Comments in the source written in a language other than English.

Resolving Header File Information

By default, the High-Level Synthesis GUI does not automatically parse all header files to resolve all coding constructs. The symptoms of this can be:

- Annotations in the code viewer which say a variable or value is unknown or cannot be defined.
- Variables in the code which do not appear in the directives window.

In both cases, the definitions for the unknown values and missing variables will be defined in a header file (a file with extension .h or .hpp). The solution to resolving the missing information is to edit the project setting using menu item Project > Project Settings... and enable the Parse All Header Files as shown in Figure 2-21.

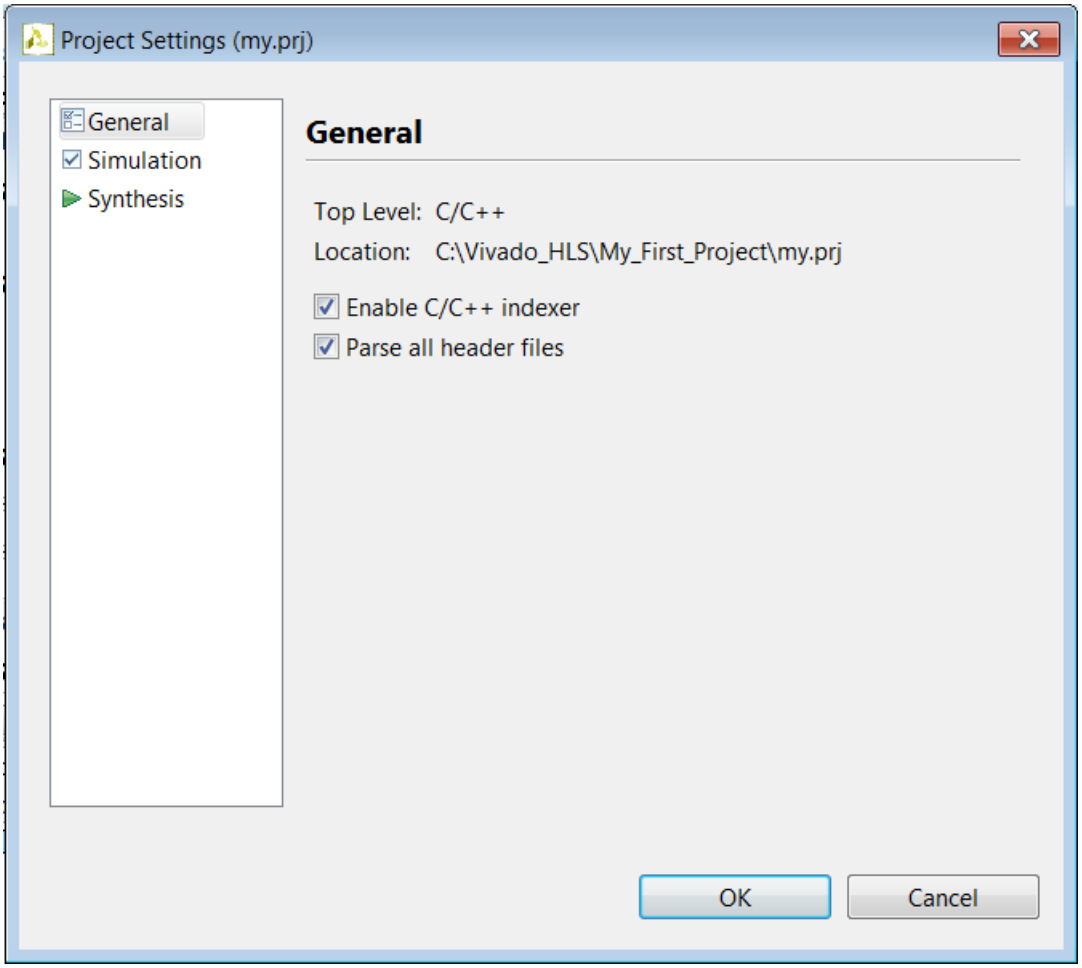

Figure 2-21: Enabling Header File Parsing

Note: When option Parse all header files is selected, the High-Level Synthesis GUI will continuously pole all header files for any potential changes. This may result in a reduced response time from the GUI as CPU cycles are used to check the header files.

Resolving Comments in the Source Code

In some localizations, non-English comments in the source file may appear as strange characters. This can be corrected by:

1. Selecting the project in the Explorer Pane.
2. Right-click and select the appropriate language encoding using Properties > Resource. In the section titled Text File Encoding select Other and choose appropriate encoding from the drop-down menu.

Customizing the GUI Behavior

The behavior of the High-Level Synthesis GUI can be customized using the menu Windows > Preferences and new user defined tool settings saved.

As an example on how detailed customizations can be performed using the Preferences menu, the following change will be made: The default setting for the key combination CTRL-TAB, is to make the active tab in the Information Pane toggle between the source code and the header file. This will be changed to make the CTRL-TAB combination make each tab in turn the active tab.

- In the Preferences menu, sub-menu General > Keys allows the Command value Toggle Source/Header to be selected and the CTRL-TAB combination removed by using the Unbind Command key.
- Selecting Next Tab in the Command column, placing the cursor in the Binding dialog box and pressing the CTRL key and then the TAB key, will cause the operation CTRL-TAB to be associated with making the Next Tab active.

Reviewing the sub-menus in the Preferences menu allows every aspect of the High-Level Synthesis GUI environment to be customized to ensure the highest levels of productivity.

Using Directives to Optimize

Directives can be used to perform various optimizations on the design. This section explains how optimizations are added to the solution. The various optimizations are discussed in detail in later chapters of this User Guide

The first step in adding optimization directives is to open the source code in the Information pane.

As shown in Figure 2-22, expand the source container, located at the top of the Explorer pane, and double-click on the source file to open it for editing in the Information pane.

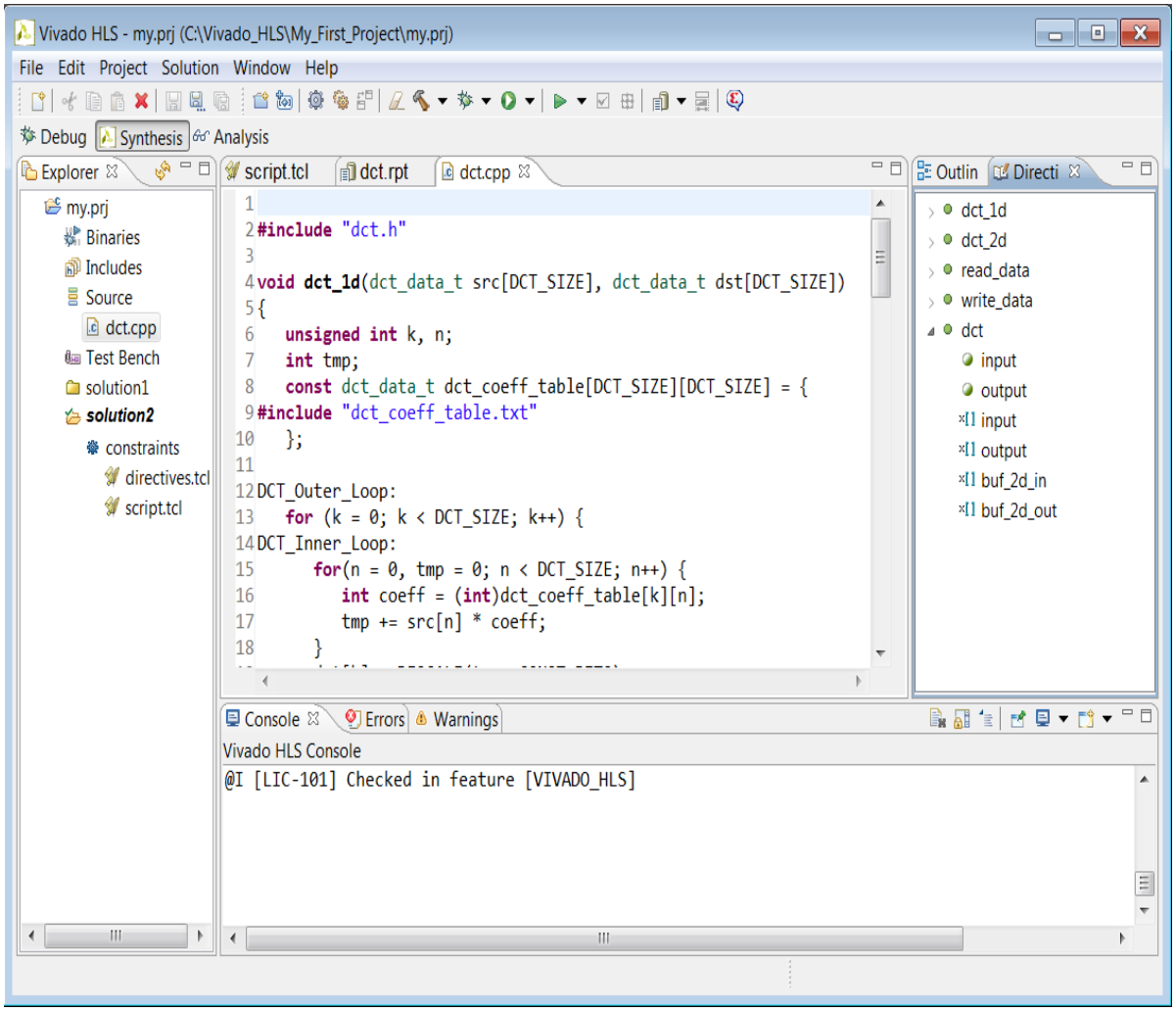

Figure 2-22: Source and Directive

With the source code active in the Information pane, the directives tab on the right-hand side becomes active.

Directives Tab

The directives tab contains all the objects in the opened source code upon which directives can be applied.

- Functions
- Interfaces
  - Interfaces are the arguments to the top-level function: these will become ports on the RTL design and directives can be specified on these to specify the IO protocol ports.

- Arrays
- Loops
- Regions
  - A region is any named region of code surrounded by braces.

Note: The objects shown in the directives tab are only those from the file currently shown in the information pane (current active file): not all files in the design.

The following example shows the outline of some source code, highlighting each of the scopes and objects upon which directives can be applied and optimizations performed.

```
int foo_sub_A (int mem_1[64],..) {
    for_A: for (int n = 0; n < 3; ++n) {
   ...
    }
    ...
}
int foo_sub_B (int mem_1[64], int i) {
    for_B:for (int n = 0; n < 4; ++n) {
   ...
    }
    ...
}
void foo_top (int mem_1[64], int mem_2[64]) {
      ...
  for_top: for (int i = 0; i < 64; ++i) {
     my_label: {
      ...
    }
  }
}
```

Figure 2-23 shows how this example code is represented in the directives tab.

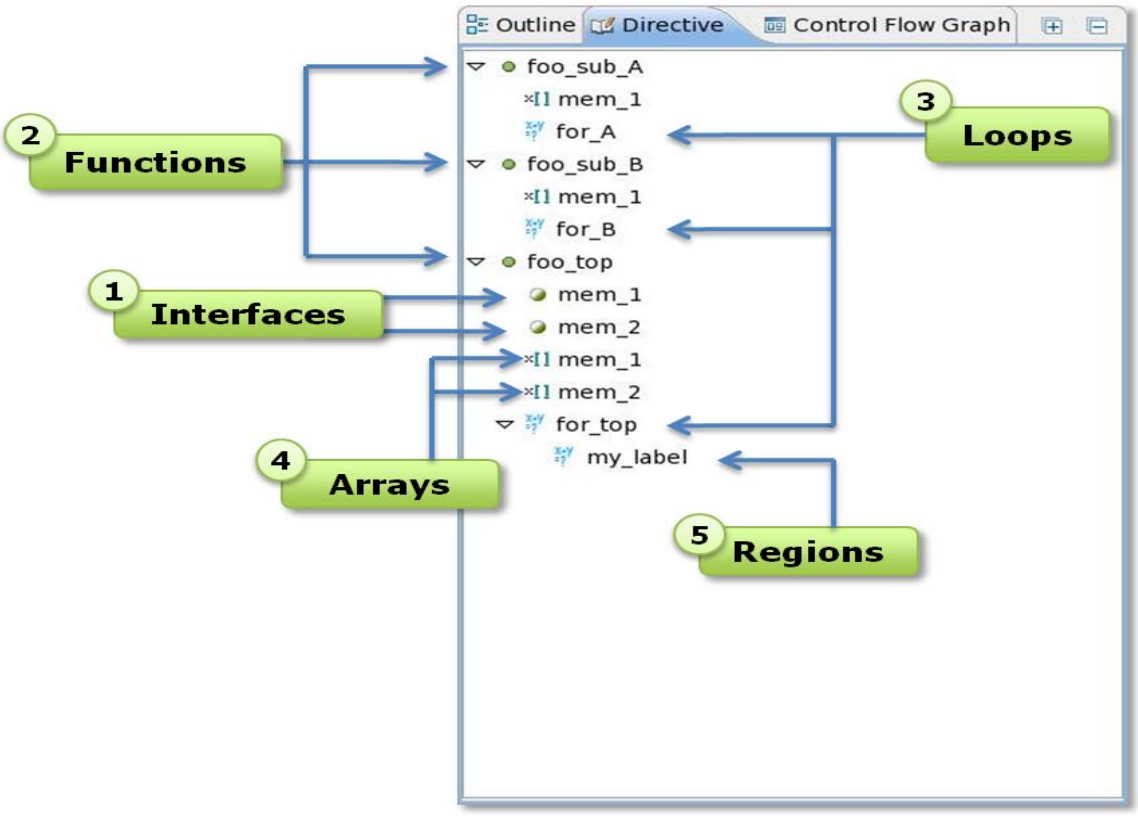

Figure 2-23: GUI Directives Objects

Applying Directives

Directives are applied by selecting an object in the directives tab and clicking with the right-hand button of mouse to open the directives window, as shown in Figure 2-24.

The drop-down menu allows the appropriate directives to be added. The example in Figure 2-24 shows the DATAFLOW directive being added. In addition to the options for the directive (discussed in later chapters) the directives window allows the directive to be inserted into the directive file as a Tcl command or to be inserted directly into the code as a pragma.

Note: To apply directives to objects in a header file, such as a class:

To add a directive to a class member or global variable, open the Directives Editor on a function that uses the variable and enter the variable name manually in Directives Editor.

To add a directive to a local scalar, open the Directives Editor on a function that contains the variable. and enter the variable manually in Directives Editor.

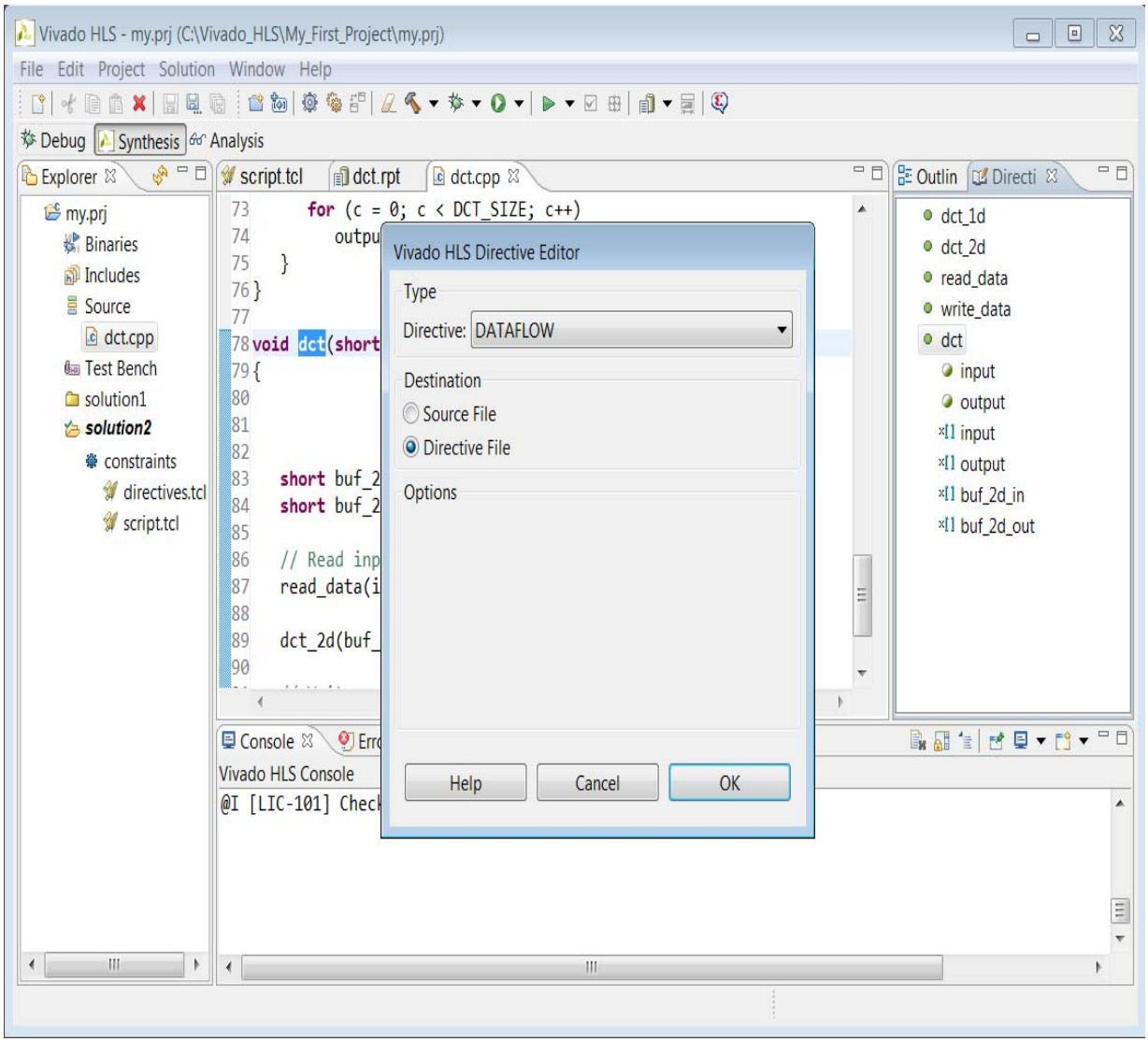

Figure 2-24: Adding Directives

When the Into Directive File option is selected in the directives dialog box, the directive is written to file directives.tcl in the solution directory. The two advantages for this approach are:

- Each solution can have its own directives.
- Users wishing to create Tcl batch files can simply copy the directive from the directives.tcl file

Figure 2-25 shows the directive being added to the directives.tcl file and shows the resulting directives.tcl opened in the information pane.

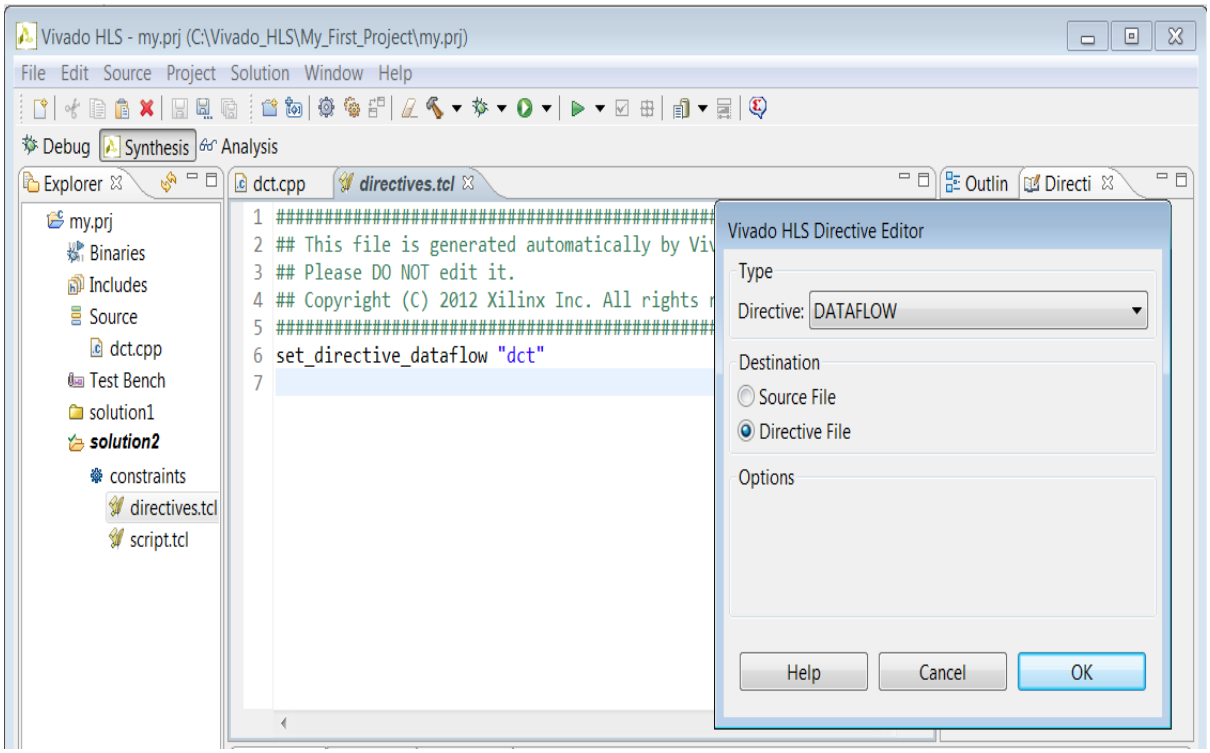

Figure 2-25: Adding Tcl Directives

The alternative option for directives is to add a pragma to the code. The advantage to this option is that the directive is permanently applied to the code and no additional files are required. This is an ideal approach for releasing IP and for directives which will never changed based on the technology target, such as the TRIPCOUNT directive.

Figure 2-26 shows the directive from the previous example being applied as a pragma, by selecting option Source File in the Destination section of Directives Editor, and the resultant source code open in the information pane.

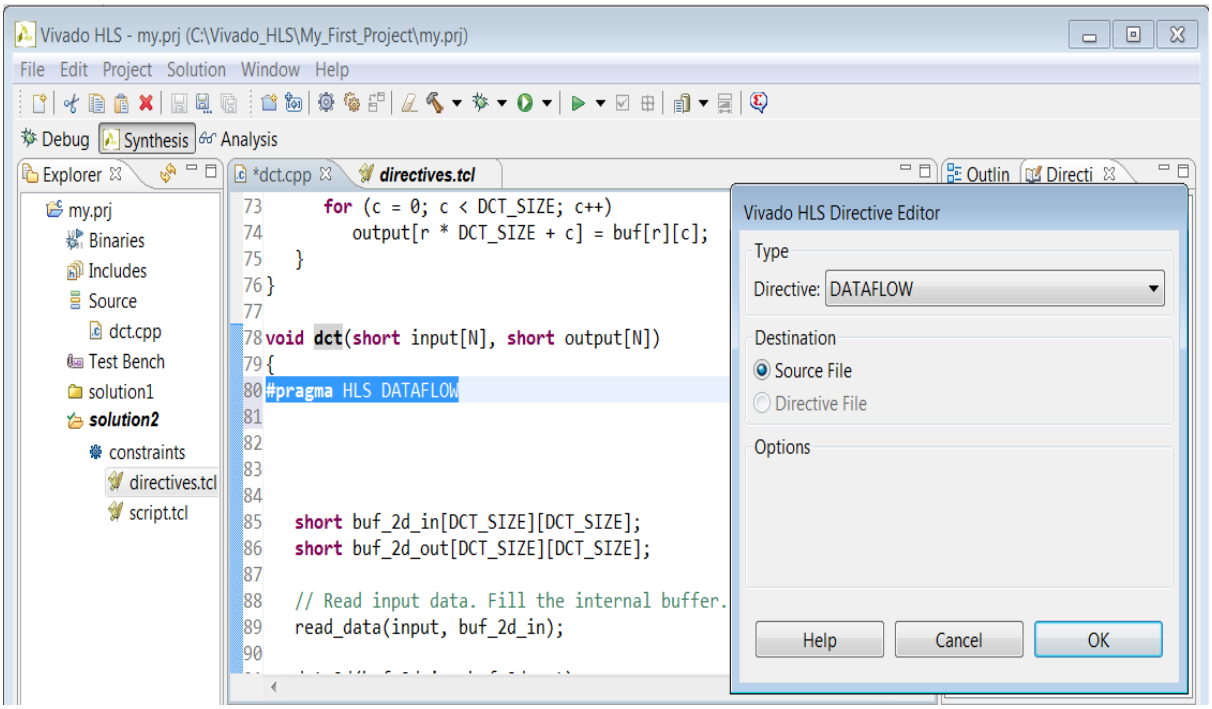

Figure 2-26: Adding Pragma Directives

In both cases, the directive will be applied and the optimization performed when synthesis is executed.

The only disadvantage to using pragmas is that the directive is now permanently embedded with the source code and will be used for every new solution.

First Example

A tutorial introduction on using High-Level Synthesis is available via the help menu.

The tutorial uses a design example to provide a good understanding of the following topics associated with using High-Level Synthesis:

- Perform validation of the C design
- Create an High-Level Synthesis project
- Perform Synthesis & Design Analysis
- Address Bit-accurate design
- Perform Design Optimization
- Understand how to perform RTL verification and export
- Review using High-Level Synthesis with Tcl scripts

The example design for use with the tutorial can be found in the examples directory in the High-Level Synthesis installation area.

C Validation and Coding Styles

Verification in an HLS flow can be separated into two discrete processes. Pre-synthesis validation which validates the C program correctly implements the required functionality and post-synthesis verification which verifies the RTL is correct. It is not uncommon for both processes to be referred to as simulation: C simulation and RTL simulation.

Pre-Synthesis Validation

Prior to synthesis, the function to be synthesized should be verified using a test bench. An ideal test bench has the following attributes:

- The test bench is self-checking.
- The test bench is in a separate file from the design (not a requirement, as discussed next, but advised).

Having the test bench and the function to be synthesized in separate files keeps a clean separation between the process of simulation and synthesis. If the test bench is in the same file as the function to be synthesized, there is a minor modification to the general High-Level Synthesis flow: the file with the test bench and the function to be synthesized should be added to the High-Level Synthesis project as a source file and as a test bench file.

Similarly, if the file with the function to be synthesized has functions above which are not in the test bench file, the file(s) with the functions above the top-level function must added to the project as a test bench file.

Typically the entire process of compiling C designs for pre-synthesis validation can be performed inside High-Level Synthesis as shown in Figure 2-17 and Figure 2-18. The C validation can however also be performed at the command line in the High-Level Synthesis Command Prompt.

C Validation outside the High-Level Synthesis GUI

Given a top-level design file "foo_top.c" and test bench file "tb_foo_top.c" the following commands can be used to compile and execute the test bench:

```
$ gcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

C++ Validation outside the High-Level Synthesis GUI

Given a top-level C++ design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench:

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
$ ./foo_top
```

SystemC Validation outside the High-Level Synthesis GUI

Given a top-level design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench (the command options for the first gcc command are shown split over multiple lines for clarity but should appear on the same command line):

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
         -I$vivado_hls_ROOT/Linux_x86_64/tools/systemc/include/
         -lsystemc
         -L$vivado_hls_ROOT/Linux_x86_64/tools/systemc/lib-linux64
$ ./foo_top
```

Since SystemC is being used the "systemc.h" header file must be included in all compilations.

Using a non-standard version of GCC

The version of gcc should be used to compile the C code prior to synthesis. High-Level Synthesis will create RTL to match the functionality of this version of gcc and this is the version which will be used to co-simulate the C test bench with the RTL.

High-Level Synthesis can be instructed to use a different version of gcc for RTL simulation by setting the environment variable AP_SIM_GCC prior to invoking High-Level Synthesis. The variable should be defined with the path to the directory which contains the local version of gcc.

Visual Studio Compiler

Microsoft Visual Studio Compiler (MVSC) can be used to compile the code prior to using High-Level Synthesis.

When the functions are to be compiled with High-Level Synthesis header files, such as those used with arbitrary precision integers (these are discussed in section “Arbitrary Precision Data Types” later in this chapter) there are special considerations to be aware of.

Compile C

C functions using arbitrary precision integers, as defined by High-Level Synthesis header file “ap_cint.h” must be compiled with APCC as discussed in the section “Arbitrary Precision Types with C”. MVSC cannot be used for C designs which use High-Level Synthesis arbitrary precision types.

Compiling C++

C++ functions which include High-Level Synthesis header files must have the location of the header file specified in MVSC.

To specify the location of the High-Level Synthesis header files in MVSC,

1. Click Project
2. Click Properties
3. In the panel that opens, select C/C++
4. Select general
5. Click on additional include directories and add the path as show in Figure 2-27.

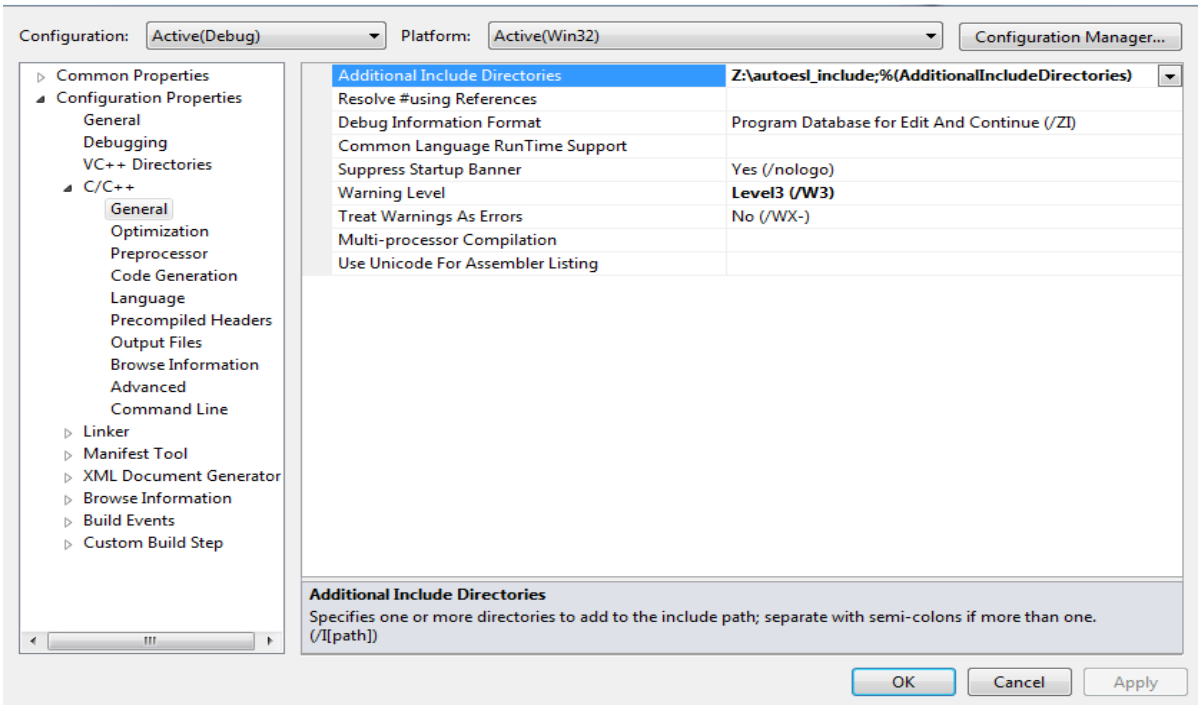

Figure 2-27: Setting High-Level Synthesis Include Path in Visual Studio

Unsupported C Language Constructs

While High-Level Synthesis is able to synthesize a large subset of all three C modeling standards (C, C++ and SystemC) there are some constructs which cannot be synthesized. This section outlines the constructs which cannot be synthesized.

In order to be synthesized, the C function must contain the entire functionality of the design (none of the functionality can be performed by system calls to the OS), the C constructs

must be of a fixed/bounded size and the implementation of those constructs unambiguous. The following constructs fail to satisfy one or more of these characteristics.

System Calls

System calls cannot be synthesized since they are actions which related to performing some task upon the operating system in which the C program is running. A few examples show how system calls cannot be synthesized into anything within the hardware design itself.

- The printf() call prints information to the system console: this is useful during C simulation but cannot be a feature of the final hardware design and as such cannot be synthesized.
- The fprintf() call accesses files in the system upon which the program is executing. Again, this cannot be performed by the final hardware: access to external data must be performed via the top-level function arguments or to global variables.

Other examples of such calls are getc(), time(), sleep() etc. all of which make calls to the operating system.

In general, most system calls cannot be synthesized. Some commonly used system calls are automatically ignored by High-Level Synthesis (e.g. printf and cout) but in general they should be removed from consideration by synthesis by using the __SYNTHESIS__ macro.

Some system calls, such as those which allocate memory for the program to access, are part of the functionality of the design and must be both removed and transformed to maintain the functionality.

Dynamic Memory & Functions

Any system calls which manage memory allocation within the system, for example malloc(), alloc() and free(), must be removed from the design code prior to synthesis.

The reason for this is that they are implying resources which are created and released during runtime: to be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

However, since they also typically used to define the functionality of the design, they must be transformed into equivalent bounded representation. The following examples show how dynamic memory allocations are transformed into equivalent bounded representations.

```
#ifndef __SYNTHESIS__
  // If synthesis is not required, use this code
  long long x = malloc (sizeof(long long));
  int* arr = malloc (64 * sizeof(int));
#else
  // For synthesis, use this code
  static long long x;
  int arr[64];
#end
```

The recommended approach is to make the above changes and re-execute the C simulation to verify the simulation with the synthesizable code is identical to the original (by adding the option -D__SYNTHESIS__ to the gcc or g++ compilation).

Similarly dynamic virtual function calls are not synthesizable. The following cannot be synthesized since it create new function at run time.

```
Class A {
public:
  virtual void bar() {…};
};

void fun(A* a) {
    a->bar();
}
A* a = 0;
if (base)
  A= new A();
else
  A = new B();

foo(a);
```

Pointer Casting

Pointer casting is not supported in the general case but is supported between native C types. The following is not synthesizable and must be transformed as shown in the example, where values are assigned using the original type.

```
struct {
  short first;
  short second;
} pair;
#ifndef __SYNTHESIS__
  // If synthesis is not required, use this code
  *(unsigned*)pair = -1U;
#else
  // For synthesis, use this code
  pair.first = -1U;
  pair.second = -1U;
#end
```

Recursive Functions

To create a hardware implementation the C function, High-Level Synthesis must be able to determine the resources which be required to implement the functionality. Recursive functions cannot be synthesize since the recursion may be endless (and have no bounds).

unsigned foo (unsigned n)

```
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Tail recursion is synthesizable. In this example, the recursion will reach a maximum limit and is therefore synthesizable.

```
unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
    if (n == 0) return m;
    return foo(n, m%n);
}
```

Standard Template Libraries

Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation. For this reason the STLs cannot be synthesized.

The solution with STLs is to create a local function with identical functionality which does not exhibit these characteristics.

Arbitrary Precision Data Types

C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL busses (corresponding to hardware) support arbitrary lengths. HLS needs a mechanism to allow the specification of arbitrary precision bit-width and not rely on the artificial boundaries of native C data types: if a 17-bit multiplier is required the user should not be force to implement this with a 32-bit multiplier.

High-Level Synthesis provides arbitrary precision data types for C, C++ and supports the arbitrary precision data types which are part of SystemC.

The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit-widths and then for the C simulation to be re-executed to validate the functionality remains identical.

The __SYNTHESIS__ macro can be used to add arbitrary precision data types to the source code while retaining the original types for reference:

```
void foo {
ifdef __SYNTHESIS__
  // use bit accurate type
  int8 a,
#else
  // Original Source
  int a,
#endif
    ...
};
```

High-Level Synthesis provides both integer and fixed point data types.

Integer Data Types

High-Level Synthesis provides arbitrary precision integer data types (Table 2-3) which manage the value of the integer numbers within the boundaries of the specified width.

Table 2-3: **Integer Data Types**

Language	Integer Data Type	Required Header
C	[u]int<precision> (1024 bits)	apcc none gcc #include "ap_cin.h"
C++	ap_[u]int<W> (1024 bits)	#include "ap_int.h"
System C	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits)	#include "systemc.h"

Arbitrary Precision Types with C

For the C language, the header file "ap_cint.h" defines the arbitrary precision integer data types [u]int. The "ap_cint.h" file is located in $VIVADO_HLS_ROOT/include (where $VIVADO_HLS_ROOT is the High-Level Synthesis installation directory).

To use arbitrary precision integer data types in a C function,

- Add header file "ap_cint.h" to the source code.
- Change the bit types to intN or uintN, where N is a bit-size from 1 to 1024.
- Compile using the apcc compiler
  - Select the Use APCC for Compiling C Compiler option in the GUI.
  - Use apcc in place of gcc at the command prompt

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include ap_cint.h

void foo_top (…) {

    int9 var1;          // 9-bit
    uint10 var2;          // 10-bit unsigned
```

Note: Standard C compilers will not correctly simulate C arbitrary precision types and the High-Level Synthesis apcc utility must be used.

Standard C compilers such as gcc will compile the attributes used in the header file to define the bit sizes, but they do know what they means. The final executable created by standard C compiler will issue messages such as the following

```
$VIVADO_HLS_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute
directive ignored
```

and proceed to use native C data types for the simulation and producing results which do not reflect the bit-accurate behavior of the code.

High-Level Synthesis includes a compiler, apcc, which overcomes this limitation and allows the function to be compiled and verified in a bit-accurate manner.

The apcc compiler can be enabled in the project setting using menu Project > Project Settings > Simulation and select Use APCC for Compiling C Files as shown in Figure 2-28.

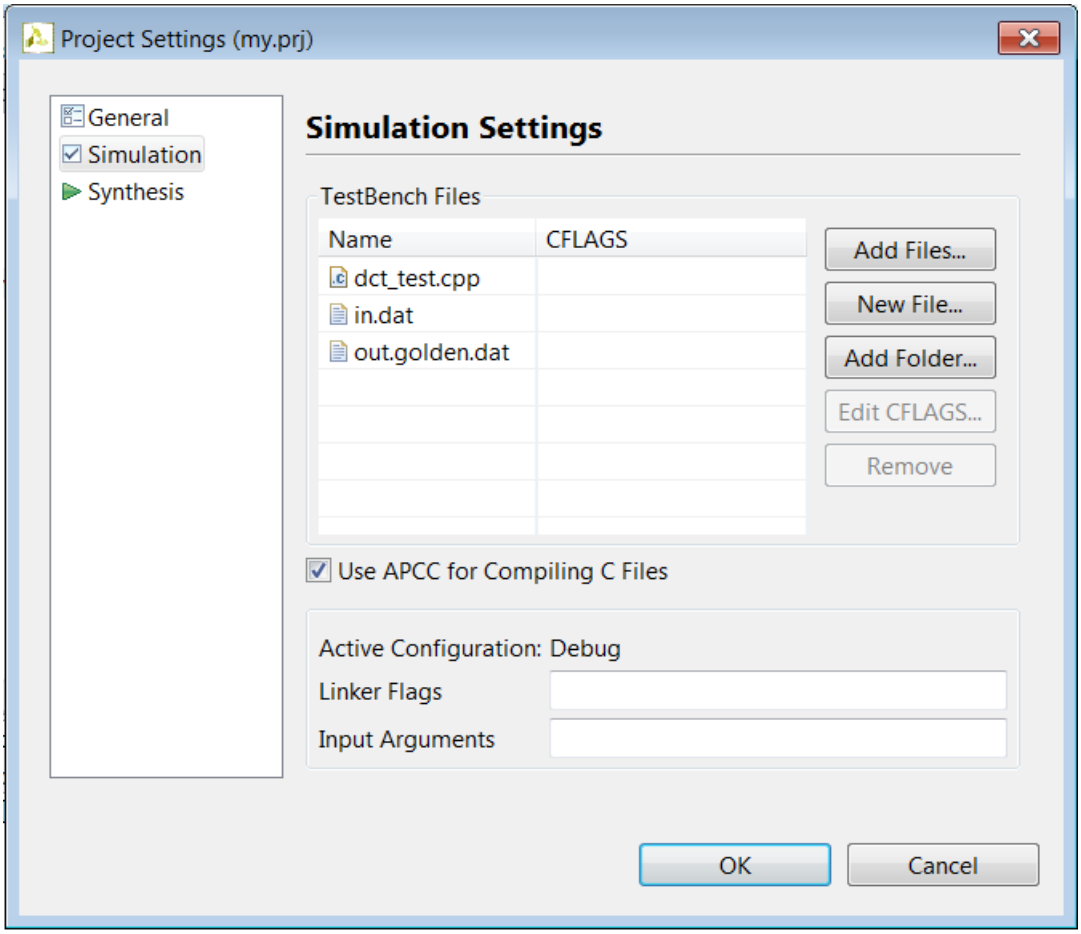

Figure 2-28: Enabling the APCC Compiler

Note: When option Use APCC for Compiling C Files is selected, the design can no longer be analyzed in the debugger: this is a side-effect of using arbitrary procession type in C code.

For functions specified using C++ or SystemC there are no such limitations when using arbitrary precision types. This limitation only exists with C, not C++ or SystemC.

APCC should not be used to compile C++ or SystemC functions (it will be ignored if selected).

If compiling at the command prompt, the apcc compiler should be used at the shell prompt: it is command line compatible with gcc and will process the arbitrary precision arithmetic correctly (respecting the boundaries imposed by the bit-width information).

When apcc is used the High-Level Synthesis header files are automatically included (no need to use –I$VIVADO_HLS_ROOT/include) and the design will simulate with the correct bit-accurate behavior.

```
$ apcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

Arbitrary Precision Types with C++

For the C++ language ap_[u]int data types, the header file “ap_int.h” defines the arbitrary precision integer data. The “ap_int.h” file is located at $VIVADO_HLS_ROOT/include (where $VIVADO_HLS_ROOT is the High-Level Synthesis installation directory).

To use arbitrary precision integer data types in a C++ function:

- Add header file "ap_int.h" to the source code.
- Change the bit types to ap_int<N> or ap_uint<N>, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and two variables have been implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include ap_int.h

void foo_top (…) {

    ap_int<9> var1;          // 9-bit
    ap_uint<10> var2;          // 10-bit unsigned
```

If arbitrary precision integers are used, the simulation must include the path to header file "ap_int.h":

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp -I$VIVADO_HLS_ROOT/include
```

Arbitrary Precision Types with SystemC

The arbitrary precision types used by SystemC are defined in the "systemc.h" header file which is required to be included in all SystemC designs. They include the SystemC sc_int<>, sc_uint<>, sc_bigint<> and sc_biguint<> types.

The path to the SystemC header file must be included when simulating SystemC designs. Given a top-level design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench on a Linux system (the command options for the first gcc command are shown split over multiple lines for clarity but should appear on the same command line):

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
        -I$VIVADO_HLS_ROOT]\Win_x86\tools\systemc\include
```

```
          -lsystemc
          -L$VIVADO_HLS_ROOT\Win_x86\tools\systemc\lib
$ ./foo_top
```

Fixed Point Data Types

The use of fixed-point types is of particular importance when using HLS since the behavior of the C++/SystemC simulations performed using fixed-point data types will match that of the resulting hardware created by synthesis, allowing analysis of the effects of bit-accuracy, quantization and overflow to be analyzed with fast C-level simulation.

High-Level Synthesis offers arbitrary precision fixed point data types (Table 2-4) for use with C++ and SystemC functions.

Table 2-4: **Fixed Point Data Types**

Language	Fixed Point Data Type	Required Header
C	-- Not Applicable --	-- Not Applicable --
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
System C	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h"

These data types manage the value of floating point numbers within the boundaries of a specified total width and integer width (Figure 2-29).

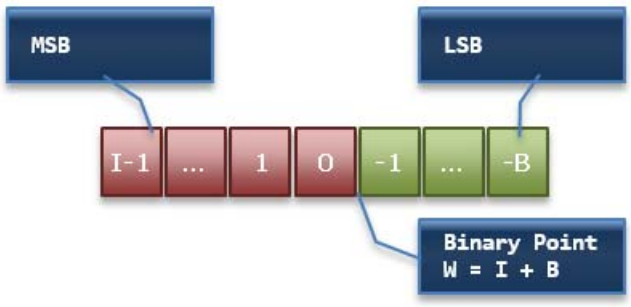

Figure 2-29: Fixed Point Data Type

Table 2-5 provides a brief overview of operations supported by fixed point types.

Table 2-5: Fixed Point Identifier Summary

Identifier	Description		
W **I**	Word length in bits The number of bits used to represent the integer value (the number of bits above the decimal point)		
Q	Quantization mode dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result. **SystemC Types:** SC_RND SC_RND_ZERO SC_RND_MIN_INF AP_RND_INF AP_RND_CONV AP_TRN AP_TRN_ZERO	**AP_Fixed Types:** AP_RND AP_RND_ZERO AP_RND_MIN_INF AP_RND_INF AP_RND_CONV AP_TRN AP_TRN_ZERO	**Description:** Rounding to plus infinity Rounding to zero Rounding to minus infinity Rounding to infinity Convergent rounding Truncation to minus infinity Truncation to zero (default)
O	The number of saturation bits in wrap modes. SystemC Types: SC_SAT SC_SAT_ZERO SC_SAT_SYM SC_WRAP SC_WRAP_SM	AP_SAT AP_SAT_ZERO AP_SAT_SYM AP_WRAP AP_WRAP_SM	Saturation Saturation to zero Symmetrical saturation Wrap around (default) Sign magnitude wrap around
N	The number of saturation bits in wrap modes.		

ap_fixed

In this example the High-Level Synthesis ap_fixed type is used to define an 18-bit variable with 6 bits representing the numbers above the decimal point and 12-bits representing the value below the decimal point. The variable is specified as signed, rounding is specified as the quantization mode and the default wrap-around mode is used for overflow.

```
#include <ap_fixed.h>
...
ap_fixed<18,6,AP_RND > my_type;
...
```

sc_fixed

In this sc_fixed example a 22-bit variable is shown with 21 bits representing the numbers above the decimal point: enabling only a minimum accuracy of 0.5. Rounding to zero is used, such that any result less than 0.5 will round to 0 and saturation is specified.

```
#define  SC_INCLUDE_FX
#define SC_FX_EXCLUDE_OTHER
#include <systemc.h>
...
sc_fixed<22,21,SC_RND_ZERO,SC_SAT> my_type;
...
```

Floating Point Types

To synthesize any design, High-Level Synthesis coverts the operations in the design into operators which are then mapped to cores from the technology library. For floating point designs not all operators have an associate floating point core in the library.

If there is no core in the technology library which can be mapped to, High-Level Synthesis synthesis will halt synthesis with a message that there is no library core to map to. A complete list of core in the High-Level Synthesis library is provided in the "High-Level Synthesis Library Guide".

Floating Point Arithmetic

In order to use a floating point core from the library, all arguments of the operation must be floating point argument. The following example code:

```
A = B/2;
```

Must be converted to:

```
A =B * 0.5;
```

In order for a floating point operator to be used (for the multiplication). High-Level Synthesis will not automatically convert the constant data type.

The standard arithmetic operations (+, -, *, / and %) are supported by floating point cores in the technology library. Simply defining a variable as a float and then using it with the standard arithmetic operators will result in a floating point core being used in the implementation.

Floating Point Math Functions

For C/C++ math functions, the function must be declared. For example, to use the sqrtf() function the following must be added to the code:

```
#include <math.h>
extern "C" float sqrtf(float);
```

The sqrtf() function can then be used with floating point variables.

Multi-Access Pointer Interfaces

Designs which use pointers in the argument list of the top-level function need special consideration when multiple accesses are performed using the pointers. Multiple accesses occur when a pointer is read from or written to, multiple times in the same function.

In the following example, (input) pointer “d_i” is read from four times and (output) “d_o” is written to twice.

```
#include "fifo.h"

void fifo ( int *d_o,  int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

When multi-access pointers are used, the following must be performed:

- It is a requirement to use the volatile qualifier.
- Specify the number of accesses on the port interface if verifying the RTL with cosim_design.
- Be sure to validate the C prior to synthesis to confirm the intent and the C model is correct

Understanding Volatile Interfaces

The code above is written with intent that input pointer "d_i" and output pointer "d_o" will be implemented in RTL as interfaces with handshakes (such as FIFO or handshake ports) which will ensure:

- Upstream producer blocks will supply new data each time a read is performed on RTL port "d_i".
- Downstream consumer blocks will accept new data each time there is a write to RTL port "d_o".

However, when this code is compiled by C compilers, the multiple accesses to each pointer will be reduced to a single access: as far as the compiler is concerned, there is no indication that the data on "d_i" changes during the execution of the function and only the final write

to "d_o" is relevant (the other writes will be over-written by the time the function completes).

High-Level Synthesis will match the behavior of the C compiler and optimize these reads and writes into a single read operation and a single write operation.

This design can be made to work as intended at the RTL by using the volatile qualifier in the code, as shown in the next example.

```
#include "fifo.h"

void fifo ( volatile int *d_o,  volatile int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

The volatile qualifier tells the C compiler, and High-Level Synthesis, to make no assumptions about the pointer accesses (the data is volatile and may change and the compiler should therefore not optimize pointer accesses).

This example results in an RTL design which will perform the expected four reads on input port "d_i" and two writes to output port "d_o".

However, even if the volatile keyword is used, this coding style (accessing a pointer multiple times) introduces two additional issues associated with the test bench and verification.

It can easily result in:

- RTL cosim_design simulation failures.
- Modeling and Simulation mismatches.

RTL cosim_design simulation failures

The following test bench can be used to validate the algorithm above. This test bench models four executions of the function, or four transactions, to highlight the operation of the code.

Note: This test bench is not self-checking: the self-checking code is omitted for clarity.

```
#include <stdio.h>

#include "foo.h"

int main () {
```

```
  int d_o, d_i;

  for (d_i=0;d_i<4;d_i++) {
    foo(&d_o,&d_i);
    printf("%d %d\n", d_i, d_o);
  }

  return 0;
}
```

The header file "foo.h" used with this example would be the following, where the function is simply declared, and allowing function "foo" to be defined in a separate file:

```
#ifndef FOO_H_
#define FOO_H_
void foo ( volatile int *d_o,  volatile int *d_i);

#endif
```

The issue with this test bench is that it only supplies a single value to the function each transaction.

In each transaction, the test bench will apply a single value, but since the volatile keyword is used, the function foo will perform four reads and hence will read the same value four times.

The test bench will validate the algorithm with the following results, showing the output is the accumulation of four input reads plus the accumulation (or output) from the previous transaction:

```
Di Do
0  0
1  4
2  12
3  24
```

When RTL verification is performed the volatile qualifier tells High-Level Synthesis to create RTL which performs four reads and if synthesized with a handshake interface, this will ensure the interface signals for new data each time port "d_i" is read.

To verify the RTL with cosim_design, High-Level Synthesis creates a SystemC wrapper with adapters around the RTL and instantiates this (C code) wrapper into the existing C test bench, as shown in Figure 2-30.

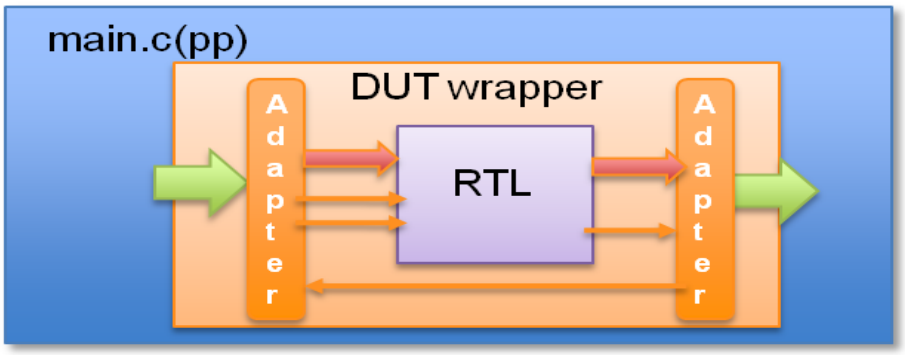

Figure 2-30: Cosim_design Wrapper Overview

The wrapper created by High-Level Synthesis models any required handshakes on the RTL interface and as such must ensure the input values to the DUT, presented by the test bench, are ready when required by the RTL design. This requires storage.

High-Level Synthesis cannot determine from this type of function interface, using pointers, how many reads or writes are performed. Neither of the arguments in the function interface tells High-Level Synthesis how many values will be read or written.

```
void foo ( volatile int *d_o,  volatile int *d_i)
```

Unless something on the interface informs High-Level Synthesis as to how many values are required, such the maximum size of an array, High-Level Synthesis will assume a single value and only create simulation wrappers for a single input and single output.

If the RTL ports are actually reading or writing multiple values, this will result in the RTL cosim_design simulation stalling: since the wrapper is modeling the producer and consumer blocks which will be connected to the RTL design, the RTL design will try to read or write the next value but the handshake interfaces will tell the design to wait, since there is currently no value to read or no space to write.

When multi-access pointers are used at the interface, High-Level Synthesis must be informed of the maximum number of reads or writes on the interface. When specifying the interface, use the depth option on the INTERFACE directive as shown in Figure 2-33.

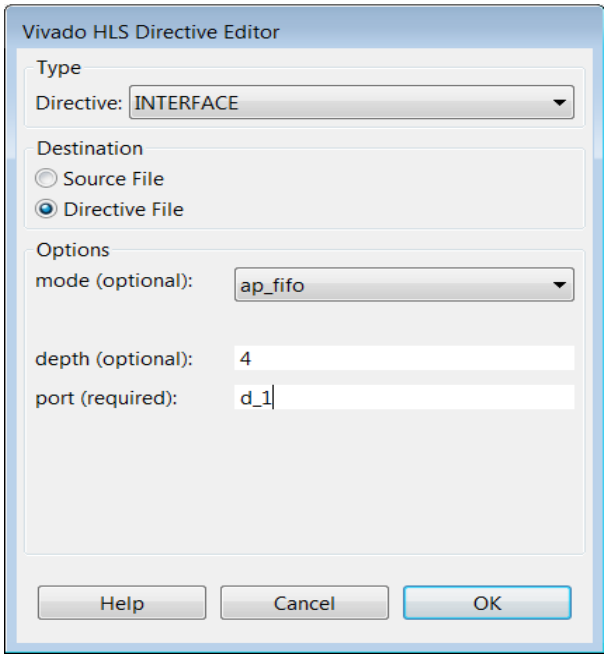

Figure 2-31: Interface Directive Dialog: Depth Option

In the above example, argument/port "d_i" is set to have a (FIFO interface) depth of 4, ensuring that cosim_design will provide enough values to correctly verify the RTL.

Simulation Mismatches and C Modeling

Once the ports are defined to have a maximum depth and cosim_design can be used to verify the RTL crated from the function, the next issue will be a simulation mismatch.

The RTL input interface will not supply the same four values in each transaction as the test bench does. At the RT level, the wrapper (modeling the RTL producer block the design will eventually be connected to) will supply a new value each time one is requested by the handshake interface. It therefore supply 4 values in the first transaction of this example: 0 the value supplied by the test bench and then three undefined values , x, y and z since the test bench only provided one value to the RTL wrapper.

There will be a simulation mismatch between the C and RTL.

Note: The issue here is the inability of this C code and test bench to correctly model multiple a situation where multiple reads or writes is required.

At the start of this example, the function was verified by reading the same four values from the test bench. This was done with the assumption that at the RT level, the data would be updated but somehow things would work out. This was the mistake.

This example was not contrived. It is something seen all too often with C function to be synthesized into hardware: the code is often written and synthesized but it does not execute as expected and a great deal of time is wasted debugging RTL.

Note: Always validate the C code with a C simulation prior to synthesis.

The limitations of this coding style can be overcome by re-writing the code.

The code shown in the next example has been updated to ensure it will read four unique values from the test bench. This is achieved by having the code access explicit values defined in the test bench. Since the pointer accesses are sequential and start at location zero, a streaming interface type can be used during synthesis.

```
#include "foo.h"

void foo ( volatile int *d_o,  volatile int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *(d_i+1);
    *d_o = acc;
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;

}
```

The test bench is updated to model the fact that the function will read four unique values in each transaction. (Note, to keep the example small and easy to understand, this new test bench only models a single transaction and the self-checking code is omitted).

```
#include "foo.h"

int main () {
  int d_o[4], d_i[4];
  int i;

  for (i=0;i<4;i++) {
    d_i[i]=i;
  }

  foo(d_o,d_i);

  printf("Di Do\n");
  for (i=0;i<4;i++) {
    if (i<2)
    printf("%d %d\n", d_i[i], d_o[i]);
    else
    printf("%d\n", d_i[i]);
  }

  return 0;
}
```

The test bench will validate the algorithm with the following results, showing there are two outputs from a single transaction and they are an accumulation of the first two input reads, plus an accumulation of the next two input reads and the previous accumulation:

```
Di Do
0 1
1 6
2
3
```

This design will synthesize and the RTL will be verified by cosim_design if the port interfaces are set to a depth of four (port "d_i") and two (port "d_o").

However, this updated example suffers from an additional limitation. Each time the function is called, it will start reading data from position 0 in the external array (d_i[0] in the test bench). This function requires that all data be available to read when the function is called

Streams can be used to work around this limitation. AP_STREAMs are a coding construct provided by High-Level Synthesis, which allows streaming data, and hence applications which use streaming data, to be more easily modeled in C.

Since, as has been seen in this example, streaming data is not easily modeled in C but is often used in hardware designs (video, communications, etc.), AP_STREAMs may be the most intuitive way to model the hardware in C. These are discussed in the "Coding with Streams" section, discussed next.

Interface Management

In C based design, all input and output operations are performed, in zero time, through formal function arguments. In an RTL design these same input and output operations must be performed via a port in the design interface and must operate using a specific IO (input-output) protocol.

High-Level Synthesis supports two solutions for specifying the type of IO protocol used:

- Interface synthesis, where the port interface is automatically created based on efficient, safe and standard interfaces.
- Manual interface specification where the interface behavior is explicitly described in the input source code. This allows any arbitrary IO protocol to be used, hence allows the function to interface with any hardware resource.
    - This solution is more typical with SystemC designs, where the IO control signals are specified in the interface declaration and their behavior specified in the code.
    - High-Level Synthesis also supports this mode of interface specification for C and C++ designs. This is detailed later in this chapter.

Interface Synthesis

When a C program is synthesized into an RTL design, the C arguments are synthesized into RTL data ports. Interface synthesis allows an interface protocol to be automatically added to the RTL data port. The interface protocol could be as simple as an output valid signal indicating when an output is ready or it could include all the ports required to interface with a BRAM, such that the date could be read from or written to a BRAM.

The type of interfaces which can be created by interface synthesis depend on the C argument. For example, for an output valid signal to be created by interface synthesis, the C argument must be a point or C++ reference, since pass-by-value scalars can only be inputs. Figure 2-32 summarizes the types of interface which are supported for each type of C function argument.

Note: Interface synthesis is not generally supported for SystemC designs.

In SystemC designs all IO protocol signals are declared in the interface declaration and their behavior is fully specified in the code.

The exception to this is for memory interfaces, as discussed in "SystemC Interface Synthesis".

If no interface type is specified for the port, the interface will be implemented with the default interface as detailed in Figure 2-32. If an unsupported interface type is specified, as also shown in Figure 2-32, High-Level Synthesis will issue a warning message and revert to the default interface type.

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I[1]	IO[2]	O[2]	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld[3]					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none[4]												
ap_ctrl_hs[4]			D									

Key:
I : input
IO : inout
O : output
D : Default Interface

Supported Interface
Unsupported Interface

Figure 2-32: Data Type and Interface Synthesis Support

The notes in Figure 2-32 are explained as follows:

1. The concept of inputs and outputs is somewhat different between C functions and RTL blocks. The following convention is used here for the purposes of explaining interface synthesis:

- A function argument which is read and never written to, like an RTL input port, is referred to as an input (I).
- A function argument which is both read and written to, like an RTL inout port, is referred to as an inout (IO)
- A function argument which is written to but never read, like an RTL output port, is referred to as an output (O)

2. A standard pass-by-value argument cannot be used to output a value to the calling function. The value of an argument such as this can only be returned (or output from the function) by the function return statement.

- Any pass-by-value function argument which is written to but never read, (like an RTL output port) will be synthesized as an RTL input port with no fanout.
- Any pass-by-value function argument which is written to and read, (like an RTL inout port) will be synthesized as an RTL input port only.

3. The ap_ovld interface type is only valid for output ports.
4. The interface types ap_ctrl_none and ap_ctrl_hs are used to control the synthesis of function level interface protocols. These interface types are specified on the function itself (all other interface types are specified on the function arguments).

Interface Types

This section details each of the interface types supported by High-Level Synthesis. The details on how to specify an interface are discussed after this section: first comes an explanation of the interfaces.

There are two distinct types on interface synthesis. Interface synthesis which is performed on C function arguments and interface synthesis which is performed at the function or block level.

Block level interface synthesis applies an IO protocol to the entire block, adding control signals to control when the block can begin operation, when it is ready for new data and when it completes. Block level synthesis is controlled by interface modes ap_ctrl_hs and ap_ctrl_none, which are applied to the function or the function return port.

Standard port level interface synthesis is specified by applying the appropriate interface mode to a function argument. A function argument which is both read from and written to (an RTL inout port) is synthesized in the following manner:

- For interface types ap_none, ap_stable, ap_ack, ap_vld, ap_ovld and ap_hs: as separate input and output ports. For example, if function argument arg1 was both read from and written to, it would be synthesized as RTL input data port arg1_i and output data port arg1_o and any specified or default IO protocol is applied to each port individually.
- For interface types ap_memory and ap_bus: a single interface is created. Both these RTL interfaces support read and write.
- For interface type ap_fifo: read and write are not supported for ap_fifo interfaces.

Structs on the interface are flattened and all hierarchy is removed before being implemented as ports: the first argument in the struct hierarchy is implemented in the LSBs of the port and the last argument implemented in the port MSBs. The implementation of arrays inside structs depends on whether the struct is a pass-by-value or pointer argument.

- In pass-by-value structs, arrays are completely scalarized with all elements inlined to create single-wide bus.
- In pointer structs, arrays ports are maintained and can be implemented in the same manner as any other array (as discussed below).

If a design is to be verified using the cosim_design feature, the following must hold true:

- The design must use block-level handshakes, as specified by ap_ctrl_hs.
- Each output port must use an interface type which indicates when a write operation has occurred: ap_vld, ap_ovld, ap_hs, ap_memory, ap_fifo or ap_bus.

The default interface types ensure the design can be verified by the cosim_design feature. SystemC designs do use interface synthesis and there are no such requirements to verify a SystemC design with cosim_design.

In the following explanations, producer blocks are those RTL blocks which provide data to the current block inputs and consumer blocks are those which consume the output data of the current block.

ap_ctrl_none & ap_ctrl_hs

Interface types ap_ctrl_none and ap_ctrl_hs are used to specify if the RTL is implemented with block-level handshake signals or not. Block-level handshake signals specify when the design can start to perform its standard operation and when that operation ends. These interface types are specified on the function or the function return.

Figure 2-33 shows the resulting RTL ports and behavior when ap_ctrl_hs is specified on a function (note, this is the default operation). In this example the function returns a value using the return statement and thus output port ap_return is created in the RTL design: if there is no function return statement this port is not created.

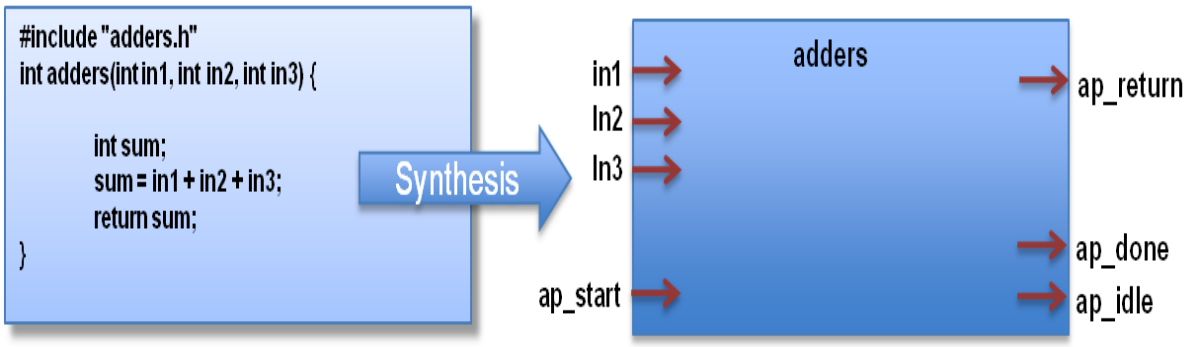

Figure 2-33: Example ap_ctrl_hs Interface

If ap_ctrl_none is specified, none of the handshake signal ports (“ap_start”, “ap_idle” and “ap_done”) shown in Figure 2-33 are created and the block cannot be verified with the cosim_design feature.

The behavior of the block level handshake signals created by interface mode ap_ctrl_hs are shown in Figure 2-34 and summarized below.

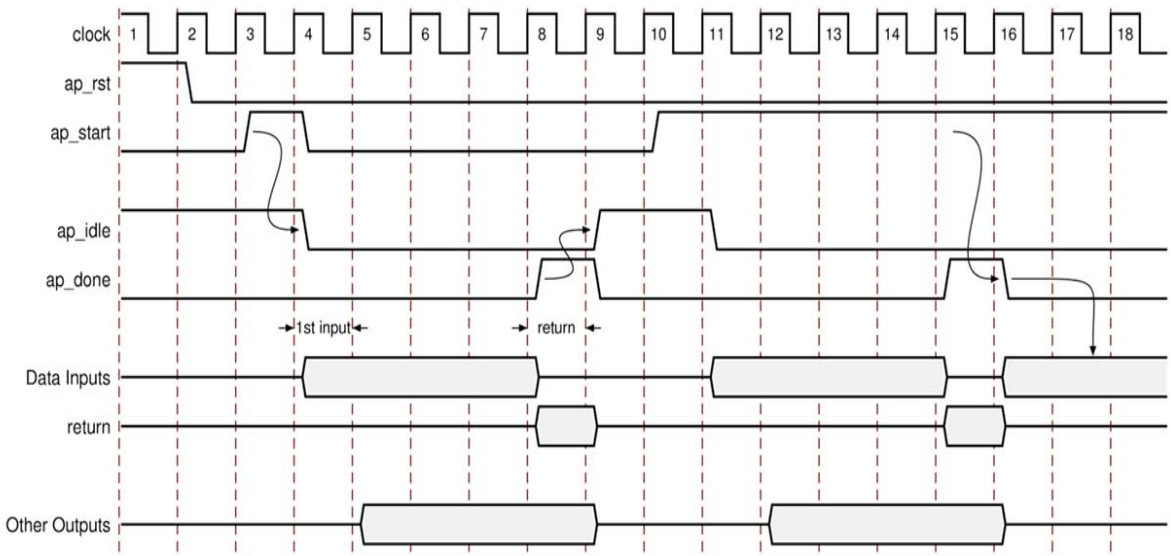

Figure 2-34: Behavior of ap_ctrl_hs Interface

After reset:

- The block will wait for “ap_start” to go high before it begins operation.
- Output “ap_idle” goes low when “ap_start” is sampled high.
- Data can now be read on the input ports.
  - The first input data may be sampled on the first clock edge after “ap_idle” goes low.
- When the block completes all operations, any return value will be written to port ap_return
  - If there was no function return, there will be no ap_return port on the RTL block.
  - Other outputs may be written to at any time until the block completes and are independent of this IO protocol.
- Output “ap_done” goes high when the block completes operation.
  - If there is an ap_return port, the data on this port will be valid when “ap_done” is high.

- The "ap_done" signal can therefore used to validate when the function return value (output on port ap_return) is valid.

- The idle signal goes high one cycle after "ap_done" and remains high until the next time "ap_start" is sampled high (indicating the block should once again begin operation).

If the "ap_start" signal is high when "ap_done" goes high:

- The "ap_idle" signal will remain low.
- The block will immediately start its next execution (or next transaction).
- The next input may be read on the next clock edge.

High-Level Synthesis supports pipelining, allowing the next transaction to begin before the current one ends. In this case, the block can accept new inputs before the first transaction completes and output port "ap_done" is asserted high.

If the function is pipelined, or if the top-level loop is pipelined with the -rewind option, an additional output port "ap_ready" is created to indicate when new inputs can be applied. Figure 2-35 shows the behavior of the block level interface when pipelining is used.

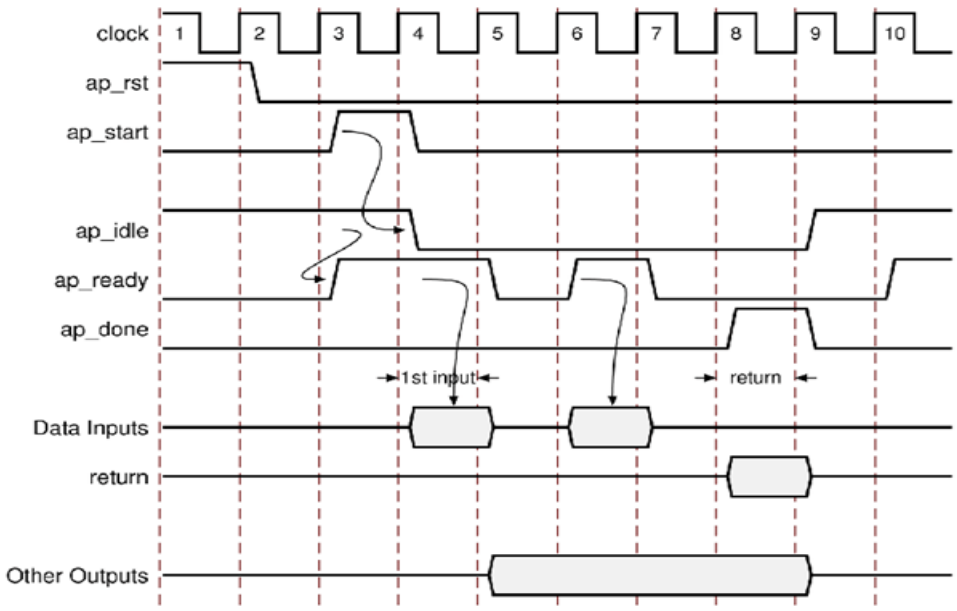

Figure 2-35: Behavior of ap_ctrl_hs Interface with Pipelining

After reset:

- If "ap_idle" is high, the "ap_ready" signal will go high.

- The block will wait for “ap_start” to go high before it begins operation.
- Output “ap_idle” goes low when “ap_start” is sampled high.
- Data will be read on the input ports when “ap_ready” is high.

Note: Since “ap_ready” can be high before “ap_idle” is low, both the ready and idle signals must be used by producer blocks for applying new data.

The remainder of the behavior is the same as that described for Figure 2-34.

Note: The only data which is guaranteed to be valid when “ap_done” is asserted is the data on the optional ap_return port (this port will only exist if a value is returned from the function using the return statement).

The other outputs in the design may be valid at this time, the end of the transaction when “ap_done” is asserted, but that is not guaranteed. If it is a requirement that an output port must have an associated valid signal, it should be specified with one of the port-level IO protocols discussed in the remainder of this section.

ap_none

The ap_none interface type is simplest interface and has no other signals associated with it. Neither the input nor output signals have any associated control ports indicating when data is read or written. The only ports in the RTL design are those specified in the source code.

An ap_none interface requires no additional hardware overhead but does require that producer blocks provide data to the input port at the correct time or hold it for the length of a transaction (until the design completes) and consumer blocks are required to read output ports at the correct time: as such, a design with interface mode ap_none specified on an output cannot be automatically verified using the cosim_design feature.

The ap_none interface cannot be used with array arguments, as shown in Figure 2-32.

ap_stable

The ap_stable interface type, like type ap_none, does not add any interface control ports to the design. The ap_stable type informs High-Level Synthesis that the data applied to this port will remain stable during normal operation, but is not a constant value which could be optimized, and the port is not required to be registered.

The ap_stable type is typically used for ports which will provide configuration data - data which may change but which will remain stable during normal operation (configuration data is typically only changed during or before a reset).

The ap_stable type can only be applied to input ports. When applied to inout ports, only the input part of the port is considered to be stable.

ap_hs (ap_ack, ap_vld and ap_ovld)

An ap_hs interface provides both an acknowledge signal to say when data has been consumed and a valid signal to indicate when data has been read. This interface type is a superset of types ap_ack, ap_vld and ap_ovld.

- Interface type ap_ack only provides an acknowledge signal.
- Interface type ap_vld only provides a valid signal.
- Interface type ap_ovld only provides a valid signal and only applies to output ports or the output half of an inout pair.

Figure 2-36 shows how an ap_hs interface behaves for both an input and output port. In this example the input port in named "in" and the output port named "out". Note how the control signals are automatically named, based on the original port name (For example, the valid port for input "in" is added and named "in_vld").

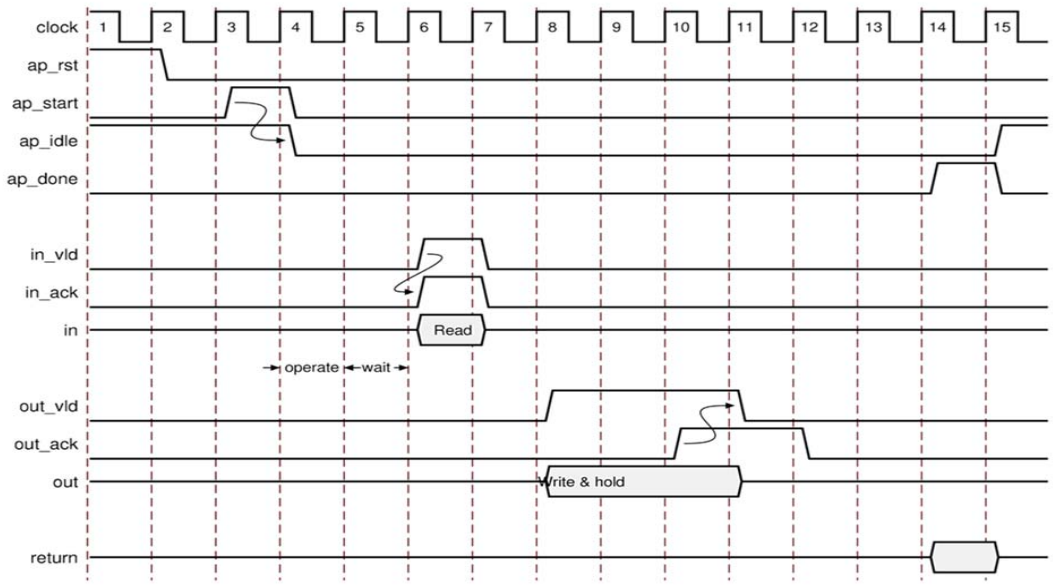

Figure 2-36: Behavior of ap_hs Interface

For inputs:

- After reset and start have been applied, the block will begin normal operation.
- If the input port is to be read but the input valid is low, the design will stall and wait for the input valid to be asserted; indicating a new input value is present.
- As soon as the input valid is asserted high, an output acknowledge will be asserted high to indicate the data was read.

For outputs:

- After reset and start have been applied, the block will begin normal operation.
- When an output port is written to, its associated output valid signal is simultaneously asserted to indicate valid data is present on the port.
- If the associated input acknowledge is low, the design will stall and wait for the input acknowledge to be asserted.
- When the input acknowledge is asserted, the output valid is de-asserted on the next clock edge.

Designs which use ap_hs interfaces can be verified with cosim_design and provide the greatest flexibility in the development process, allowing both bottom-up and top-down design flows: all intra-block communication is safely performed by two-way handshakes, with no manual intervention or assumptions required for correct operation.

The ap_hs interface is a safe interface protocol but requires a two port overhead, with associated control logic.

With an ap_ack interface, only an acknowledge port is synthesized.

- For input arguments this results in an output acknowledge port which is active high in the cycle in which the input is read.
- For output arguments this results in an input acknowledge port.
  - After a write operation, the design will stall and wait until the input acknowledge has been asserted high, indicating the output has been read by a consumer block.
  - However, there is no associated output port to indicate when the data can be consumed.

Care should be taken when specifying output ports with ap_ack interface types. Designs which use ap_ack on an output port cannot be verified by cosim_design.

Specifying interface type ap_vld results in a single associated valid port in the RTL design.

- For output arguments this results in an output valid port, indicating when the data on the output port is valid.

  Note: For input arguments this valid port behaves in a different manner than the valid port implemented with ap_hs.
- If ap_vld is used on an input port (there is no associated output acknowledge signal), the input port will be read as soon as the valid is active: even if the design is not ready to read the port, the data port will be sampled and held internally until needed.
- The input data will be read each cycle the input valid is active.

An ap_ovld interface type is the same an ap_vld but can only be specified on output ports. This is a useful type for ensuring pointers which are both read from and written to, will only be implemented with an output valid port (and the input half will default to type ap_none).

ap_memory

Array arguments are typically implemented using the ap_memory interface. This type of port interface is used to communicate with memory elements (RAMs, ROMs) when the implementation requires random accesses to the memory address locations. Array arguments are the only arguments which support a random access memory interface.

If sequential access to the memory element is all that is required, the ap_fifo interface discussed next can be used to reduce the hardware overhead: no address generation is performed in an ap_fifo interface.

When using an ap_memory interface, the array targets should be specified using the set_directive_resource command as shown in section "Memory Resource Selection". If no target is specified for the arrays, High-Level Synthesis will automatically determine if a single or dual-port RAM interface will be used.

TIP: *Ensure array arguments are targeted to the correct memory type using RESOURCE directive before synthesis as re-synthesizing with new, correct, memories may result in a different schedule and RTL.*

Figure 2-37 shows an example where an array named “d” has the resource specified as a single-port BRAM.

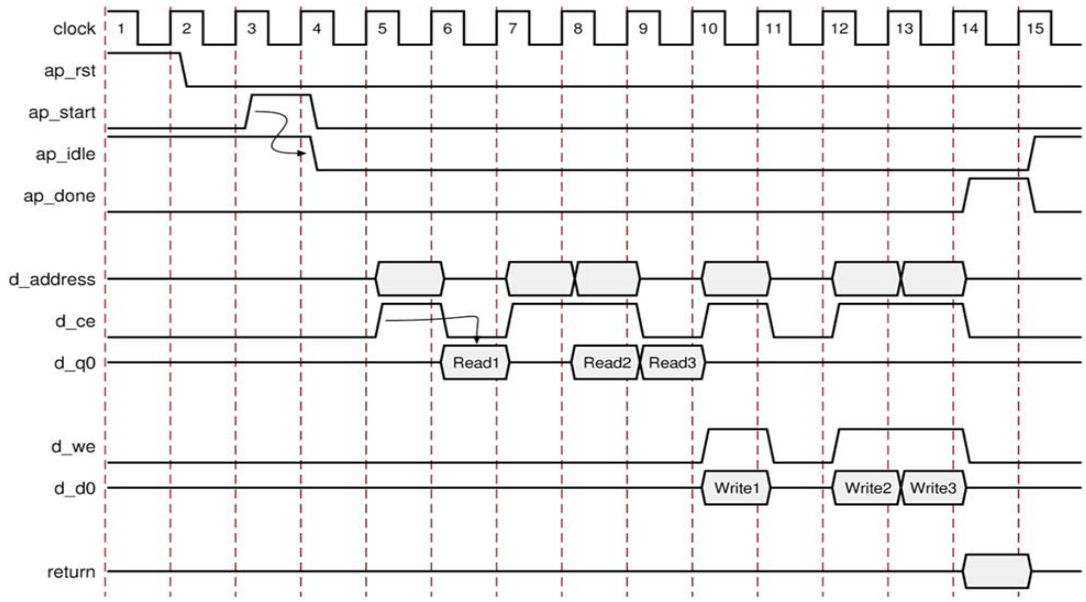

Figure 2-37: Behavior of ap_memory Interface

After reset:

- After reset and start have been applied, the block will begin normal operation.
- Reads will be performed by applying an address on the output address ports while asserting the output signal "d_ce".
  - For this BRAM target, the design expects the input data to be available in the next clock cycle.
- Write operations will be performed by asserting output ports "d_ce" and "d_we" while simultaneously applying the address and data.

A memory interface cannot be stalled by external signals, provides an indication of when output data is valid and can therefore be verified using cosim_design.

ap_fifo

If access to a memory element is required and the access is only ever performed in a sequential manner (no random access required) an ap_fifo interface is the most hardware efficient. The ap_fifo interface allows the port to be connected to a FIFO, supports full two-way empty-full communication and can be specified for array, pointer and pass-by-reference argument types.

Functions which can use an ap_fifo interface will often use pointers and may access the same variable multiple times. Refer to "Multi-Access Pointer Interfaces" to understand the importance of the volatile qualifier when this coding style is used.

Note: An ap_fifo interface assumes that all reads and writes are sequential in nature.

If High-Level Synthesis can determine this is not the case, it will issue an error and halt.

If High-Level Synthesis cannot determine that the accesses are always sequential, it will issue a warning that it is unable to confirm this and proceed.

In the following example "in1" is a pointer which accesses the current address, then two addresses above the current address and finally one address below.

```
void foo(int* in1, ...) {
  int data1, data2, data3;
        ...
  data1= *in1;
  data2= *(in1+2);
  data3= *(in1-1);
  ...
}
```

If "in1" is specified as an ap_fifo interface, High-Level Synthesis will check the accesses and determine the accesses are not in sequential order, issue an error and halt. To read from non-sequential address locations use an ap_memory interface as this random accessed or use an ap_bus interface.

An ap_fifo interface cannot be specified on an argument which is both read from and written to (an inout port). Only input and output arguments can be specified with an ap_fifo interface. An interface with input argument "in" and output argument "out" specified as ap_fifo interfaces will behave as follows:

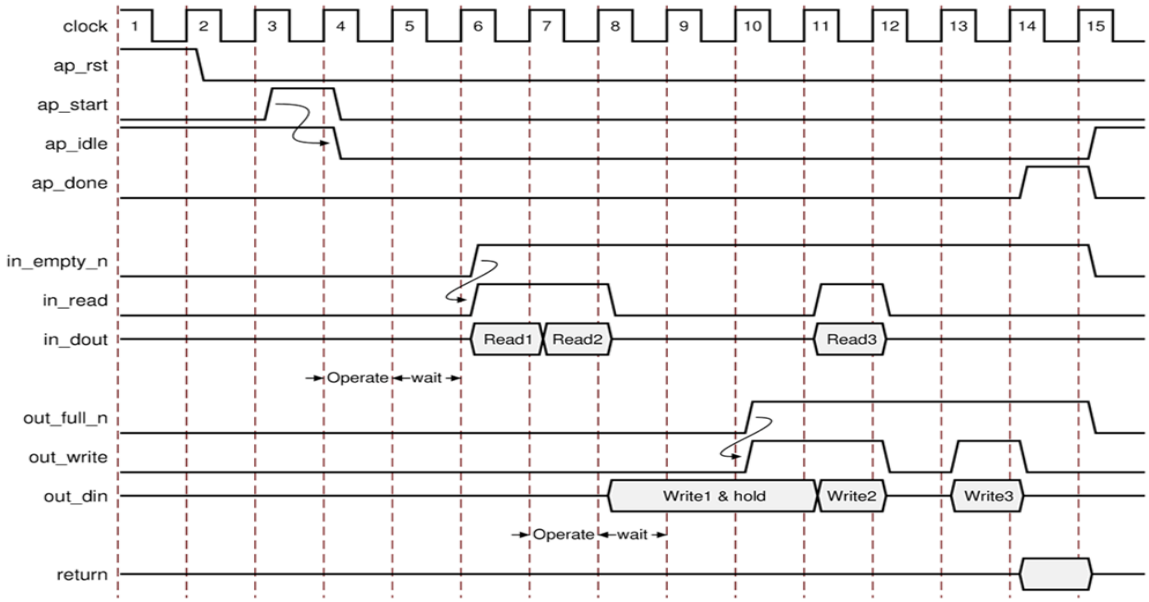

Figure 2-38: Behavior of ap_fifo Interface

After reset and start have been applied, the block will begin normal operation.

For reads:

- If the input port is to be read but the FIFO is empty, as indicated by input port "in_empty_n" low, the design will stall and wait for data to become available.
- As soon as the input port "in_empty_n" is asserted high to indicate data is available, an output acknowledge ("in_read") is asserted high to indicate the data was read in this cycle.
- If data is available in the FIFO when a port read is required, the output acknowledge will be asserted to acknowledge data was read in this cycle.

For outputs:

- If an output port is to written to but the FIFO is full, as indicated by "out_full_n" low, the data will be placed on the output port but the design will stall and wait for the space to become available in the FIFO.
- When space becomes available in the FIFO (input "out_full_n" goes high) the output acknowledge signal "out_write" is asserted to indicate the output data is valid.
- If there is space available in the FIFO when an output is written to, the output valid signal is asserted to indicate the data is valid in this cycle.

In an ap_fifo interface the output data port has an associated write signal: ports with an ap_fifo interface can be verified using cosim_design.

ap_bus

An ap_bus interface can be used to communicate with a bus bridge. The interface does not adhere to any specific bus standard but is generic enough to be used with a bus bridge which in-turn arbitrates with the system bus. The bus bridge must be able to cache all burst writes.

Functions which can employ an ap_bus interface will often use pointers and may access the same variable multiple times. Refer to "Multi-Access Pointer Interfaces" to understand the importance of the volatile qualifier when this coding style is used.

An ap_bus interface can be used in two specific ways.

- Standard Mode: The standard mode of operation is to perform individual read and write operations, specifying the address of each.
- Burst Mode: If the C function memcpy is used in the C source code, burst mode will be used for data transfers. In burst mode, the base address and the size of the transfer is indicated by the interface: the data samples are then quickly transferred in consecutive cycles.

Figure 2-39 and Figure 2-40 show the behavior for read and write operations in standard mode, when an ap_bus interface is applied to argument “d” in the following example:

```
void foo (int *d) {
  static int acc = 0;
  int i;

  for (i=0;i<4;i++) {
    acc += d[i+1];
    d[i] = acc;
  }
}
```

While Figure 2-41 and Figure 2-42 show the behaviors when the C memcpy function and burst mode is used, as in this example code:

```
void bus (int *d) {
  int buf1[4], buf2[4];
  int i;

  memcpy(buf1,d,4*sizeof(int));

  for (i=0;i<4;i++) {
    buf2[i] = buf1[3-i];
  }

  memcpy(d,buf2,4*sizeof(int));
}
```

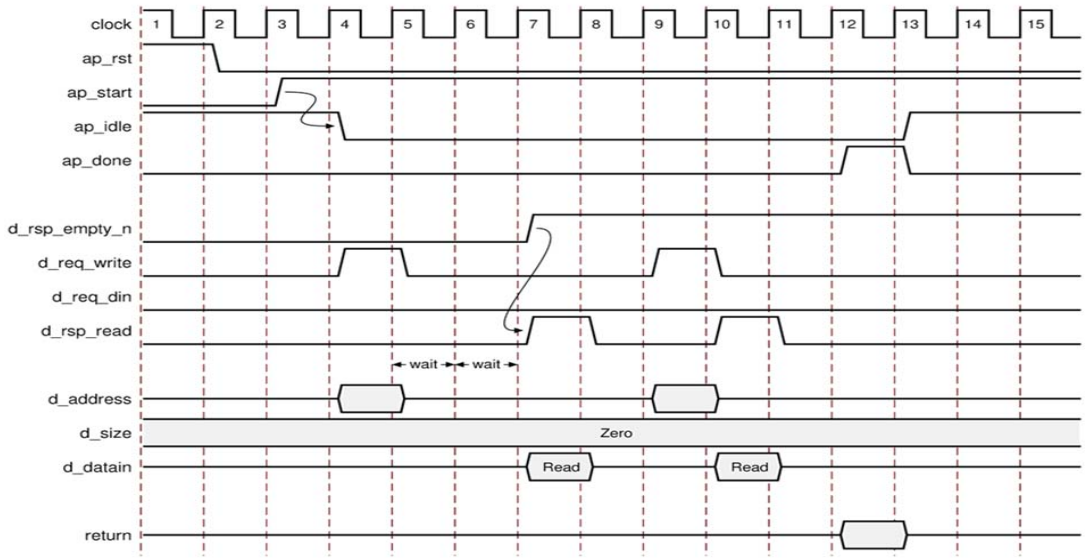

Figure 2-39: Behavior of ap_bus Interface: Standard Read

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a read is to be performed, but there is no data in the bus bridge FIFO ("d_rsp_empty_n" is low):
  - Output port "d_req_write" is asserted with port "d_req_din" de-asserted, to indicate a read operation.
  - The address is output.
  - The design will stall and wait for data to become available.
- When data becomes available for reading the output signal "d_rsp_read" is immediately asserted and data is read at the next clock edge.
- If a read is to be performed, and data is available in the bus bridge FIFO ("d_rsp_empty_n" is high):
  - Output port "d_req_write" is asserted and port "d_req_din" is de-asserted, to indicate a read operation.
  - The address is output.
  - "d_rsp_read" is asserted in the next clock cycle and data is read at the next clock edge.

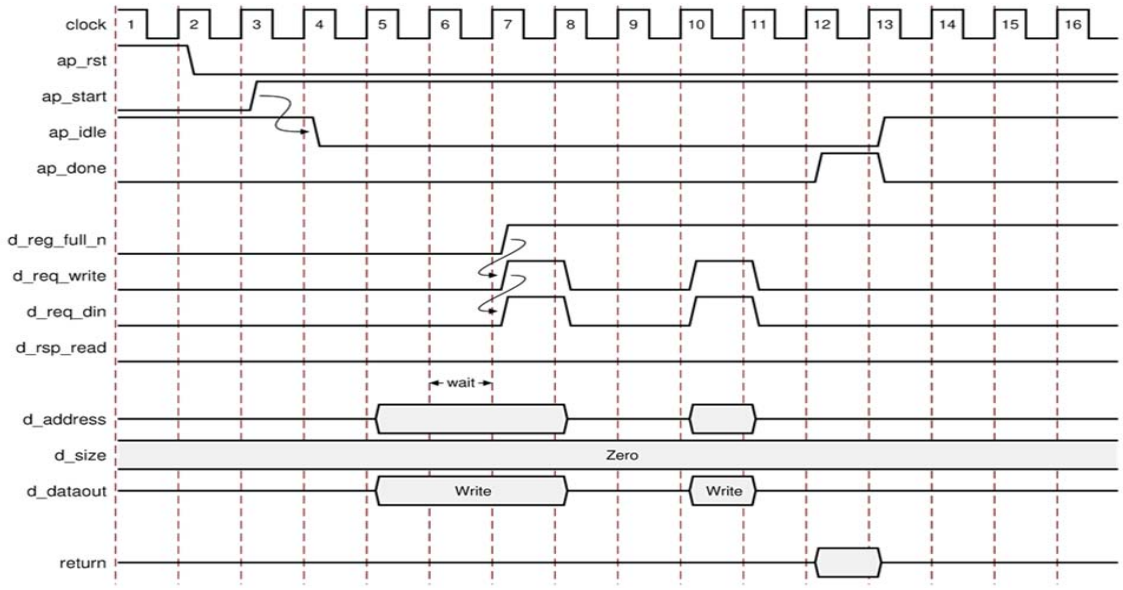

Figure 2-40: Behavior of ap_bus Interface: Standard Write

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a write is to be performed, but there is no space in the bus bridge FIFO ("d_req_full_n" is low):
  - The address and data are output.
  - The design will stall and wait for space to become available.
- When space becomes available for writing:
  - Output ports "d_req_write" and "d_req_din" are asserted, to indicate a write operation.
  - The output signal "d_req_din" is immediately asserted to indicate the data is valid at the next clock edge.
- If a write is to be performed, and space is available in the bus bridge FIFO ("d_rsp_full_n" is high):
  - Output ports "d_req_write" and "d_req_din" are asserted, to indicate a write operation.
  - The address and data are output.
  - "d_req_din" is asserted to indicate the data is valid at the next clock edge.

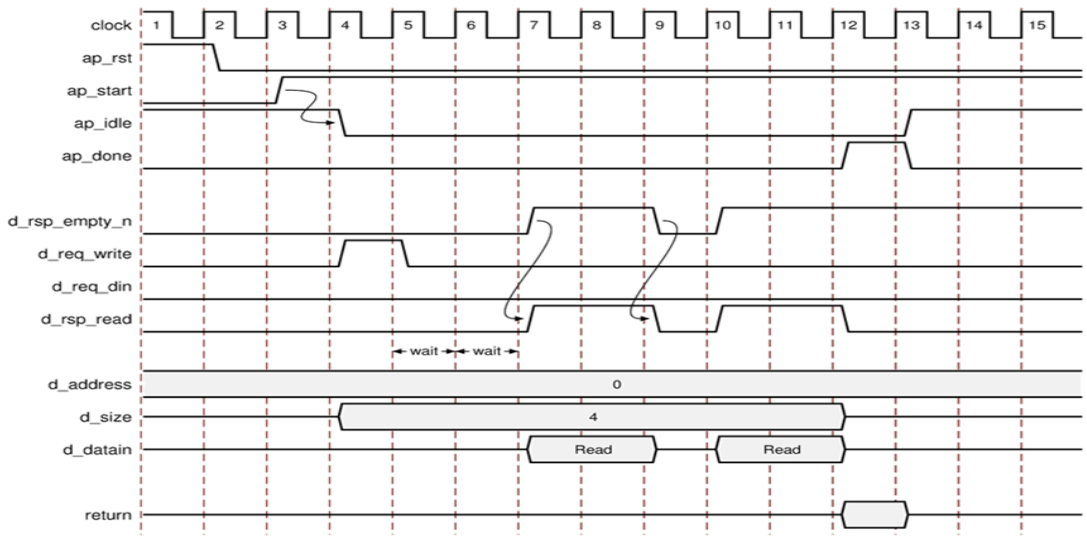

Figure 2-41: Behavior of ap_bus Interface: Burst Read

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a read is to be performed, but there is no data in the bus bridge FIFO ("d_rsp_empty_n" is low):
  - Output port "d_req_write" is asserted with port "d_req_din" de-asserted, to indicate a read operation.
  - The base address for the transfer and the size are output.
  - The design will stall and wait for data to become available.
- When data becomes available for reading the output signal "d_rsp_read" is immediately asserted and data is read at the next N clock edges, where N is the value on output port size.
- If the bus bridge FIFO runs empty of values, the data transfers stop immediately and wait until data is available before continuing where it left off.
- If a read is to be performed, and data is available in the bus bridge FIFO
  - Transfer begin as above and the design will stall and wait if the FIFO empties

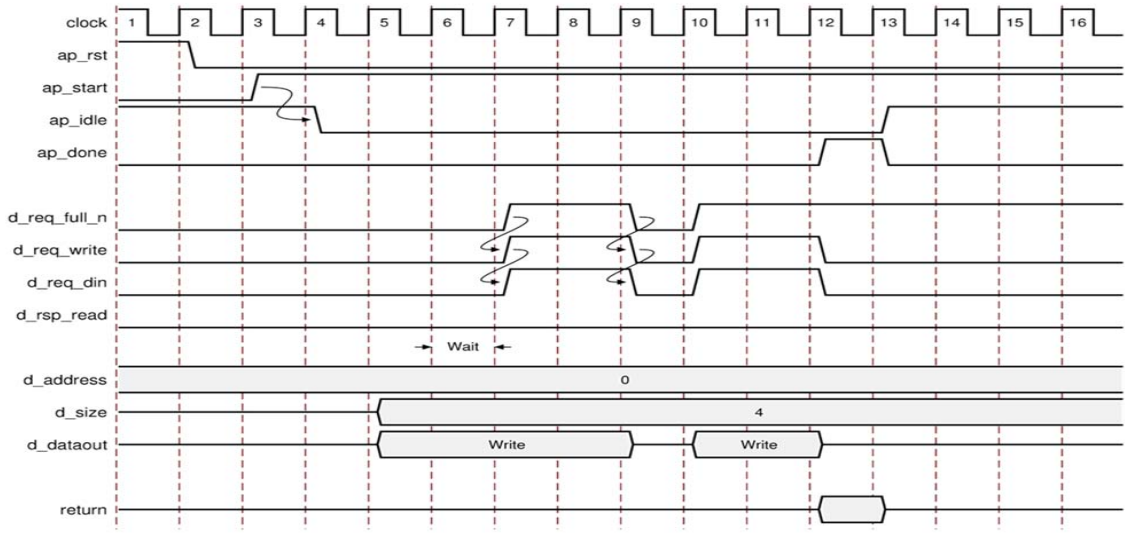

Figure 2-42: Behavior of ap_bus Interface: Burst Write

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a write is to be performed, but there is no space in the bus bridge FIFO ("d_req_full_n is low):
  - The base address, transfer size and data are output.
  - The design will stall and wait for space to become available.
- When space becomes available for writing:
  - Output ports "d_req_write" and "d_req_din" are asserted, to indicate a write operation.
  - The output signal "d_req_din" is immediately asserted to indicate the data is valid at the next clock edge.
  - Output signal "d_req_din" will be immediately de-asserted if the FIFO becomes full and re-asserted when space is available.
  - The transfer will stop after N data values have been transferred, where N is the value on the size output port.
- If a write is to be performed, and space is available in the bus bridge FIFO ("d_rsp_full_n" is high):
  - Transfer begins as above and the design will stall and wait when the FIFO is full.

The ap_bus interface can be verified by cosim_design.

Controlling Interface Synthesis

Interface synthesis is controlled by the INTERFACE directive or by using a configuration setting.

Configuration settings can be used to specify the default operation for creating RTL ports and interfaces. The INTERFACE directive is used to specify the explicit interface type of a particular port and overrides any default or global configuration.

Configuring Default Port Interface Types

Configuration settings can be used to:

- Add a global clock enable to the RTL design.
- Create RTL ports for any global variables.
- Set a default interface type for all ports of the specified type.

The configuration can be set using the GUI option Solution > Solution Settings... > General > config_interface, as shown in Figure 2-43.

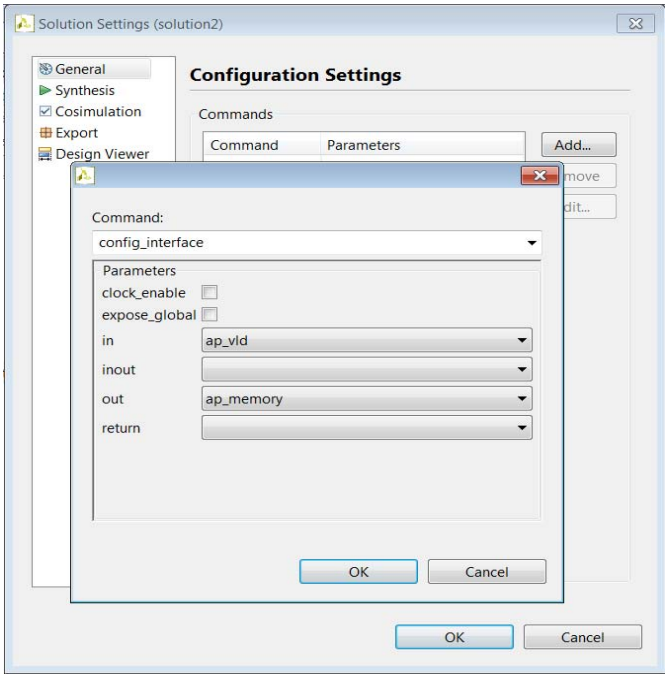

Figure 2-43: Setting a Configuration

The interface configuration setting clock_enable adds global clock enable (CE) to be added to the RTL design when selected. This adds port ap_ce to the top-level ports: when port ap_ce is active low, the clock is inhibited to all registers in the design.

Any C function can use global variables: those variables defined outside the scope of any function. By default, global variables do not result in the creation of RTL ports: High-Level Synthesis will assume the global variable is inside the final design. The configuration setting expose_global tells High-Level Synthesis the global variable will actually be implemented outside the scope of the function/design and to create an RTL port.

Finally, the default interface type for all ports was shown in Figure 2-32. The interface configuration allows these default interfaces to be specified by the user. The following example, shown in Figure 2-43 and shown below using the associated Tcl command, sets the interface type on all input ports to type ap_vld and the default type on all output ports to type ap_memory.

```
config_interface -mode in ap_vld
config_interface -mode out ap_memory
```

Note: If a port is specified with an unsupported interface type, the configuration will be ignored and the interface will be set to the default type for that port: the unsupported port types and the default port types are shown in Figure 2-32.

For example, if an array port is specified to be implemented with an unsupported IO protocol type such as ap_ack, the directive or configuration will be ignored.

Specifying Port Interface Types

The type of interface can be set on specific ports. Simply select the port in the GUI directives tab and right-click on the mouse to open the directives menu as shown in Figure 2-44.

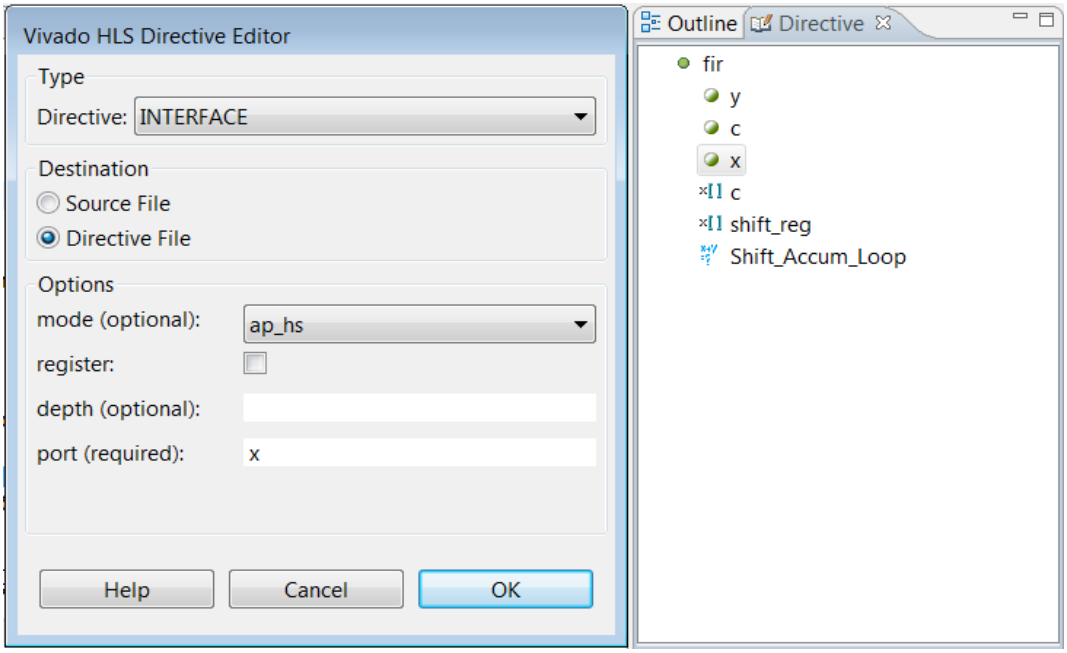

Figure 2-44: Specifying Port Interfaces

The interface can also be specified in Tcl command file. In this example, the same operation as Figure 2-44 is specified and port x in function foo is set to type ap_hs.

```
set_directive_interface -mode ap_hs fir x
```

If an unsupported port interface type is selected, a message will be issued and the default port type as shown in Figure 2-32 will be implemented.

Specifying Port Options

The INTERFACE directive has two options. The first specifies if the port should be registered or not (the default) and second specifies the number of values read from or written to the port.

By default High-Level Synthesis does not register ports. It will read input ports when the data is required saving the area overhead of a register. If the register option is selected High-Level Synthesis will register all pass-by-value reads in the first cycle of operation. For pointer read it will register the read but it will perform the read one cycle before the data is required. For output ports, the register option will guarantee the output is registered. For memory, FIFO and bus interfaces the register option has no effect.

For cases where a pointer is read from or written to multiple times within a single transaction, the depth option should be used to specify the maximum number of values read or written. Failure to specify this correctly may results in a verification failure when the cosim_design feature is used to verify the RTL.

Specifying Bus Interfaces

In addition to the standard interfaces explained in the Interface Synthesis section, Vivado HLS can also automatically add bus interfaces to the RTL design.

The primary difference between bus interface ports and the RTL ports created by interface synthesis (ap_none, ap_hs, etc) is that the bus interfaces are added to the design during the Export RTL process. Details on the RTL export process are provided later in the chapter Exporting the RTL Design.

As such:

- Bus interfaces are not reported in the synthesis reports.
- Bus interfaces are not present in the RTL written after synthesis.

The following bus interfaces are available:

- AXI4 Lite Slave
- AXI4 Master
- AXI4 Stream
- PLB 4.6 Slave
- PLB 4.6 Master
- FSL
- NPI

Another important aspect of bus interfaces is that the type of bus interface which can be used depends on the protocol of RTL port (ap_none, ap_hs, ap_bus etc). Each type of RTL interface can only be connected to certain bus interfaces.

Table 2-6 shows a list of the RTL interface ports Vivado HLS creates and bus interfaces which can be connected to them. For example, an AXI4 Stream bus interface can only be added to ports of type `ap_fifo`.

Table 2-6: RTL Port to Bus Interface Mappings

	Bus Interface Protocol						
RTL Interface Protocol	**AXI4 Lite Slave**	**AXI4 Master**	**AXI4 Stream**	**PLB 4.6 Slave**	**PLB 4.6 Master**	**FSL**	**NPI**
ap_bus	-	X	-	-	X	-	-
ap_fifo	-	-	X	-	-	X	-

Table 2-6: RTL Port to Bus Interface Mappings

Bus Interface Protocol							
RTL Interface Protocol	**AXI4 Lite Slave**	**AXI4 Master**	**AXI4 Stream**	**PLB 4.6 Slave**	**PLB 4.6 Master**	**FSL**	**NPI**
ap_ctrl_hs **ap_none** **ap_vld** **ap_ack** **ap_hs**	X	-	-	X	-	-	-
ap_ovld **ap_memory**	-	-	-	-	-	-	-

For example, if the RTL port protocol is of type `ap_fifo`, it can be connected to an AXI4 Stream or FSL bus interface. If on the other hand, the RTL port is of type `ap_ovld` or `ap_memory`, it cannot be connected to any bus interface:

- An ap_memory interface does not require an interface and can be directly connected to memories (BRAM) in EDK.
- Any port with ap_ovld interface should be modified to be one of the supported types, for example ap_hs, or it cannot be connected to an interface.

Adding a Bus Interface

To add a bus interface onto existing RTL interface, a resource is specified on the port using the RESOURCE directive. (The same process and directive is used to specify which type of memory resource is connected to an array port).

The RESOURCE directive requires that a core be specified. A complete list of all bus interface and their corresponding cores is provided in the "Vivado HLS Library Guide" and is shown in Table 2-7.

Table 2-7: Bus Interface Interfaces

Bus Interface	Core	Description
AXI4 Lite Slave	AXI4LiteS	AXI4 slave interface
AXI4 Master	AXI4M	AXI4 Master interface
AXI4 Stream	AXI4Stream	AXI4 Steam interface
PLB 4.6 Slave	PLB46S	Standard bus-slave interface.
PLB 4.6 Master	PLB46M	Standard bus-master interface.
FSL	FSL	Standard Xilinx FSL interface.
NPI	NPI64M	Native multi-port memory controller interface.

The processing of creating a bus interface is therefore a two-step process:

- Create an RTL interface.
- Specify a bus interface resource.

The following example shows how port "out" can be connected to an AXI4 Master bus interface.

Selecting an RTL Interface

The first step in creating a bus interface is to create an RTL port which supports the required bus interface. Since an AXI4 Master bus interface is required, Table 2-7 shows that the RTL port must be implemented with an `ap_bus` protocol.

In Figure 2-45, function argument "out" is selected and a directive is applied by right-clicking with the mouse. After selecting INTERFACE from the drop-down menu, the interface mode is specified as `ap_bus`.

The other options in the directive dialog allow the port to be optionally registered and the required depth to be specified for RTL verification (if the port is a pointer which is accessed multiple times, the number of accesses must be specified here).

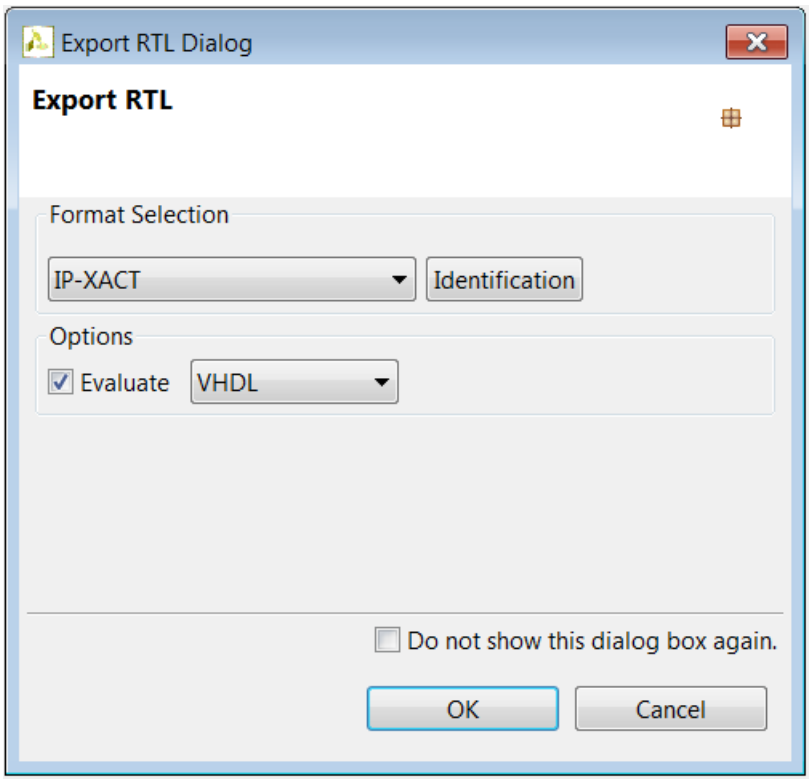

Figure 2-45: RTL Interface Directive

Selecting a bus interface

To specify an AXI4 Master bus interface, the RESOURCE directive is used as shown in Figure 2-46 to add resource AXI4M to port/argument out.

The options associated with the RESOURCE directive allows certain bus interfaces to contain multiple ports and allow use of side-channel signals in AXI4 interfaces.

- When adding AXI4 Lite Slave or PLB 4.6 Slave interfaces, the port map option allows multiple ports to be grouped into a single slave port. Use of this option is discussed with these bus interfaces.
- The metadata option is used to map specific variables in the C function to specific signals in the AXI4 interface standard. Examples of this are shown below for AXI4 Stream and AXI4 Master interfaces.

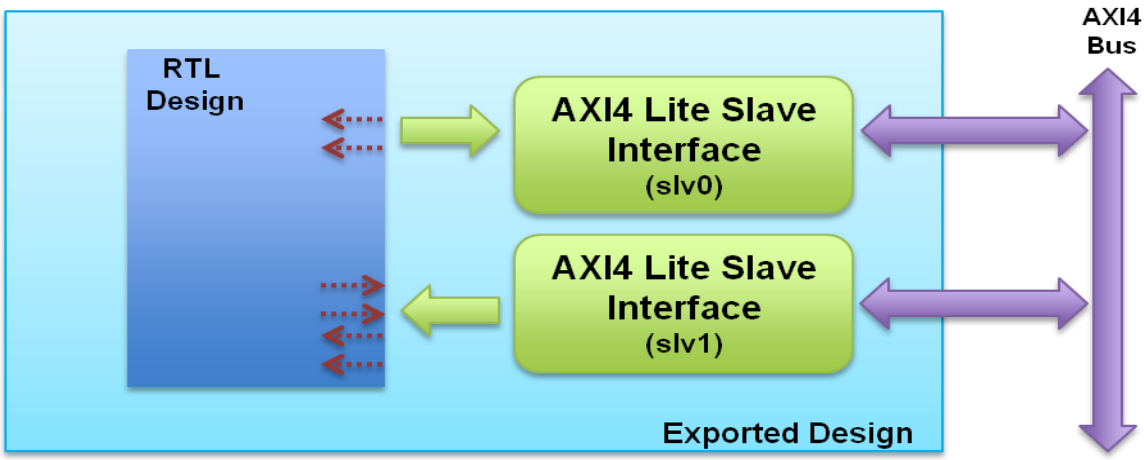

Figure 2-46: Bus Interface Resource Directive

AXI4 Lite Slave Interface

An AXI4 slave interface is typically used to allow the design to be controlled by some form of CPU or microcontroller. As such, it provides features only available in slave ports:

- Multiple RTL ports can be grouped into the same bus interface.
- When the design is exported as a Pcore for the EDK environment, the output includes C function and header files for use with the code running on the processor.

The following example shows how multiple RTL ports are bundled into a common AXI4 slave interface, allowing multiple RTL ports to be accessed through a single bus interface, and reviews the C files created.

```
int foo_top (int *a, int *b, int *c, int *d) {
```

```
// Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_none port=b
#pragma AP interface ap_vld port=c
#pragma AP interface ap_ack port=d
#pragma AP interface ap_ctrl_hs port=return register

// Define the pcore interfaces and group into AXI4 slave "slv0"
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=b

// Define the pcore interfaces and group into AXI4 slave "slv1"
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=return
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=c
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=d

    *a += *b;
    return (*c + *d);
}
```

In the example above, ports "a" and "b" are grouped into a common AXI4 Lite Slave interface as shown in Figure 2-47.

Block-level IO protocol is explicitly set by applying interface mode `ap_ctrl_hs` to the function `return`. (This is the default but shown explicitly in this example). Grouping the `return` port with ports "c" and "d", means all the block level IO protocol signals (`ap_start`, `ap_done`, etc) are assigned to the AXI4 Lite Slave interface slv1.

If the RESOURCE directive is applied using the GUI, the entire bus bundle string used to create the groupings (e.g. "-bus_bundle slv1") including the quotes should be entered into the metadata option box. Use the same bus_bundle name e.g. slv1, for each RTL port to be grouped.

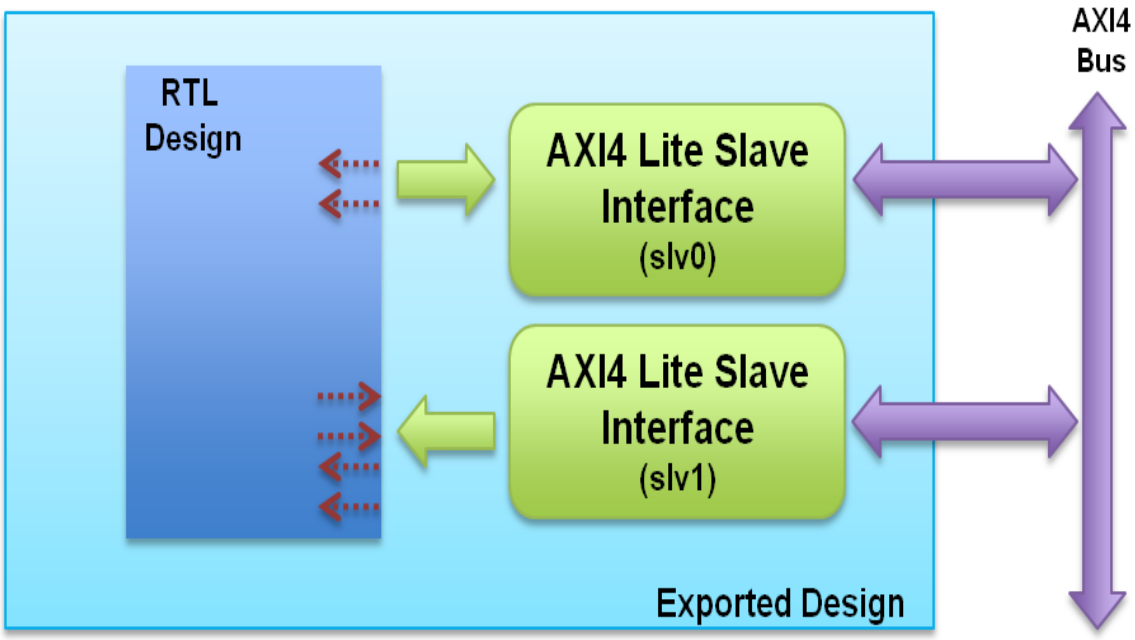

Figure 2-47: AXI4 Lite Slave Interfaces with Grouped RTL Ports

When the design is exported as a Pcore, the following C files are output in addition to the RTL design, Refer to the chapter Exporting the RTL Design (the files and filenames shown are for this example):

- xfoo_top_slv0.h
- xfoo_top_slv1.h
- xfoo_top.h
- xfoo_top.c

The file xfoo_top_slv0.h provides the memory mapped locations of the RTL ports grouped in interface slv0. The comments in the file show how the RTL port data and control signals are mapped into the AXI4 bus interface.

- Port "a" is read-write signal and so it implemented as separate input and output ports.
- Port "a" has both a valid and acknowledge signal associated with it.
- Port "b" is type ap_none and has no associated control signal.

```
// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Control signal of a_i
//        bit 0  - a_i_ap_vld (Read/Write/COH)
//        bit 1  - a_i_ap_ack (Read)
//        others - reserved
```

```
// 0x14 : Data signal of a_i
//        bit 31~0 - a_i[31:0] (Read/Write)
// 0x18 : Control signal of a_o
//        bit 0  - a_o_ap_vld (Read)
//        bit 1  - a_o_ap_ack (Read/Write/COH)
//        others - reserved
// 0x1c : Data signal of a_o
//        bit 31~0 - a_o[31:0] (Read)
// 0x20 : reserved
// 0x24 : Data signal of b
//        bit 31~0 - b[31:0] (Read/Write)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
Handshake)

#define XFOO_TOP_SLV0_ADDR_A_I_CTRL 0x10
#define XFOO_TOP_SLV0_ADDR_A_I_DATA 0x14
#define XFOO_TOP_SLV0_BITS_A_I_DATA 32
#define XFOO_TOP_SLV0_ADDR_A_O_CTRL 0x18
#define XFOO_TOP_SLV0_ADDR_A_O_DATA 0x1c
#define XFOO_TOP_SLV0_BITS_A_O_DATA 32
#define XFOO_TOP_SLV0_ADDR_B_DATA   0x24
#define XFOO_TOP_SLV0_BITS_B_DATA   32
```

File xfoo_top_slv1.h provides similar information, however, since interface “slv1” includes the block level IO protocol signals (associated with the function return) it also includes in details on setting and controlling the block interrupt.

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/SC)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        others - reserved
// 0x10 : Control signal of c
//        bit 0  - c_ap_vld (Read/Write/SC)
//        others - reserved
// 0x14 : Data signal of c
//        bit 31~0 - c[31:0] (Read/Write)
// 0x18 : Control signal of d
//        bit 1  - d_ap_ack (Read/COR)
//        others - reserved
// 0x1c : Data signal of d
//        bit 31~0 - d[31:0] (Read/Write)
// 0x20 : Data signal of ap_return
//        bit 31~0 - ap_return[31:0] (Read)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
Handshake)

#define XFOO_TOP_SLV1_ADDR_AP_CTRL   0x00
#define XFOO_TOP_SLV1_ADDR_GIE       0x04
```

```
#define XFOO_TOP_SLV1_ADDR_IER       0x08
#define XFOO_TOP_SLV1_ADDR_ISR       0x0c
#define XFOO_TOP_SLV1_ADDR_C_CTRL    0x10
#define XFOO_TOP_SLV1_ADDR_C_DATA    0x14
#define XFOO_TOP_SLV1_BITS_C_DATA    32
#define XFOO_TOP_SLV1_ADDR_D_CTRL    0x18
#define XFOO_TOP_SLV1_ADDR_D_DATA    0x1c
#define XFOO_TOP_SLV1_BITS_D_DATA    32
#define XFOO_TOP_SLV1_ADDR_AP_RETURN 0x20
#define XFOO_TOP_SLV1_BITS_AP_RETURN 32
```

Each of the address locations noted in the interface files above, have an associated C function to access it. These functions are provided in file `xfoo_top.c` and defined in `xfoo_top.h`, as shown next:

```
int XFoo_top_Initialize(XFoo_top *InstancePtr, XFoo_top_Config *ConfigPtr);

void XFoo_top_Start(XFoo_top *InstancePtr);
u32 XFoo_top_IsDone(XFoo_top *InstancePtr);
u32 XFoo_top_IsIdle(XFoo_top *InstancePtr);
u32 XFoo_top_GetReturn(XFoo_top *InstancePtr);

void XFoo_top_SetC(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetC(XFoo_top *InstancePtr);
void XFoo_top_SetCVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetCVld(XFoo_top *InstancePtr);
void XFoo_top_SetD(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetD(XFoo_top *InstancePtr);
u32 XFoo_top_GetDAck(XFoo_top *InstancePtr);
void XFoo_top_SetA_i(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetA_i(XFoo_top *InstancePtr);
void XFoo_top_SetA_iVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_iVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_iAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_o(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_oVld(XFoo_top *InstancePtr);
void XFoo_top_SetA_oAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_oAck(XFoo_top *InstancePtr);
void XFoo_top_SetB(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetB(XFoo_top *InstancePtr);

void XFoo_top_InterruptGlobalEnable(XFoo_top *InstancePtr);
void XFoo_top_InterruptGlobalDisable(XFoo_top *InstancePtr);
void XFoo_top_InterruptEnable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptDisable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptClear(XFoo_top *InstancePtr, u32 Mask);
u32 XFoo_top_InterruptGetEnabled(XFoo_top *InstancePtr);
u32 XFoo_top_InterruptGetStatus(XFoo_top *InstancePtr);
```

The names of these functions will vary depending upon the design name and the types of ports in the design. This example includes every type of port and is useful for explaining the general case. The functions created for this example are explained here, in the order they would typically be called in the C program used to control and access the block via the AXI4 Lite Slave interface.

For example, the block is first instantiated in the program space, then the interrupts are set, the block is started, port values are accessed and the results are read at the end.

First, use this function to instantiate an instance of `XFoo_Top` in the program space:

```
int XFoo_top_Initialize(XFoo_top *InstancePtr, XFoo_top_Config *ConfigPtr);
```

If interrupts are to be used, they must first be enabled in the processor. Refer to the documentation for the processor for details on performing this task. If the interrupts are not enabled in the processor the interrupt functions provided here will have no effect.

As shown later, it is possible to poll the device rather than use an interrupt service routine. The following functions are however provided for use with an interrupt service routine. There are two interrupts supported in the AXI Lite Slave interface:

- The first is status of the `ap_done` signal and available is in location 1.
- The second is a task level interrupt and is available in location 2. The task level interrupt shows when a new task can be started and allows operations on the port to be pipelined: a new task can be started before the `ap_done` signal/interupt indicates the first task is finished.

These functions will (in this order) allow the following interrupt service tasks to be performed in the C code:

- Return the status of all interrupt sources.
- Show which interrupts are enabled.
- Allow individual interrupts to be cleared.

```
u32 XFoo_top_InterruptGetEnabled(XFoo_top *InstancePtr);
u32 XFoo_top_InterruptGetStatus(XFoo_top *InstancePtr);
void XFoo_top_InterruptClear(XFoo_top *InstancePtr, u32 Mask);
```

The next task is to enable the interrupts. The global interrupt function enables all interrupts from the hardware block.

```
void XFoo_top_InterruptGlobalEnable(XFoo_top *InstancePtr);
```

The individual interrupts can be enabled using the following function. A mask value of 1 enables an interrupt from the `ap_done` signal; a value of 2 enables task level interrupt; a value of 3 enables both types of interrupt.

```
void XFoo_top_InterruptEnable(XFoo_top *InstancePtr, u32 Mask);
```

At this stage, before starting the block, any ports which do not use an input handshake (e.g. ap_none or ap_ack) should be configured (else, as soon as the block is started it will read the default values in the interface registers). In this example, port “b” and “d” have such protocols. The following functions can be used to set the value on these ports:

```
void XFoo_top_SetB(XFoo_top *InstancePtr, u32 Data);
```

```
void XFoo_top_SetD(XFoo_top *InstancePtr, u32 Data);
```

Although not a requirement, the following functions can be used to read back the data from ports "b" and "d" to verify the correct data was written.

```
u32 XFoo_top_GetB(XFoo_top *InstancePtr);
u32 XFoo_top_GetD(XFoo_top *InstancePtr);
```

The block can be started by issuing a call to the start function (first, it may be worth checking the status of the idle signal to confirm the last process is indeed finished):

```
u32 XFoo_top_IsIdle(XFoo_top *InstancePtr);
void XFoo_top_Start(XFoo_top *InstancePtr);
```

Input ports which use handshakes can have their values set before the block is started, or after the block is started. If the block is already started, but there is no valid value on an input, the block will stall until valid data is present.

These writes should be performed in a similar manner to the hardware protocol: write the data value, then set the valid bit to specify the data is valid. When the data is read by the block, the valid bit in the adapter is automatically cleared.

In this example, ports "a" (input side) and "c" are being set after the block has been started. The following functions can be used to set these values.

```
void XFoo_top_SetA_i(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetA_iVld(XFoo_top *InstancePtr);

void XFoo_top_SetC(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetCVld(XFoo_top *InstancePtr);
```

In a similar manner to ports "b" and "d", the values on ports "a" and "c" can be read back using the following functions to confirm they are correct.

```
u32 XFoo_top_GetA_i(XFoo_top *InstancePtr);
u32 XFoo_top_GetC(XFoo_top *InstancePtr);
```

In addition, the status of the valid flags can be read: remember, if the block has already been started the valid flag will be automatically cleared as soon as the data is read by the hardware.

```
u32 XFoo_top_GetA_iVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetCVld(XFoo_top *InstancePtr);
```

In this example, port "a" and port "d" have output acknowledge signals associated with their IO protocol. The status of the output acknowledges can also be read: if this is high, the data read has been acknowledged by the hardware.

The following functions can be used to obtain the status of the acknowledge ports – as soon as these registers are read by the function call, the register bit is cleared and can only be set by the hardware.

```
u32 XFoo_top_GetA_iAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetDAck(XFoo_top *InstancePtr);
```

In this example, while the block is running port "a" (output side) and the return port (ap_return) are the only ports which will return data. The following function can be used to check if there is any valid output data on port "a":

```
u32 XFoo_top_GetA_oVld(XFoo_top *InstancePtr);
```

Once valid data is present on the output port, it can be read:

```
u32 XFoo_top_GetA_o(XFoo_top *InstancePtr);
```

If the read is successful, the following function can be used to acknowledge the read. Remember, the hardware will stall until an output write is acknowledged.

```
void XFoo_top_SetA_oAck(XFoo_top *InstancePtr);
```

Once the hardware has read acknowledge register in the AXI Lite Slave interface, it will automatically clear the acknowledge register. This can be confirmed using function:

```
u32 XFoo_top_GetA_oAck(XFoo_top *InstancePtr);
```

Finally, with all other ports serviced the return value from the function can be read. If no interrupts are being used, the device can be polled to check if it is done:

```
u32 XFoo_top_IsDone(XFoo_top *InstancePtr);
```

Whether polling for the ap_done signal, or using an interrupt service routine, the following function can be used read the return value:

```
u32 XFoo_top_GetReturn(XFoo_top *InstancePtr);
```

Finally, if interrupts are being used, the interrupts for the block can be disabled (individually and/or globally).

```
void XFoo_top_InterruptDisable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptGlobalDisable(XFoo_top *InstancePtr);
```

AXI4 Master Interface

To create an AXI4 Master interface, the RTL port must have an ap_bus interface, as shown in Table 2-7. This example sets the port "m" as an ap_bus and then specifies that the port connect to an "AXI4M" resource.

The block-level interface protocol is removed in this example by setting the IO protocol to ap_ctrl_none. This ensures there are no other interface signals in this example (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256
```

```
typedef uint32 DT;

void foo_top (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as an AXI4 master
#pragma AP resource core=AXI4M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

When the pcore is generated, an AXI4 Master interface will be connected to port "m" and this can connect to an AXI4 system bus, as shown in Figure 2-48.

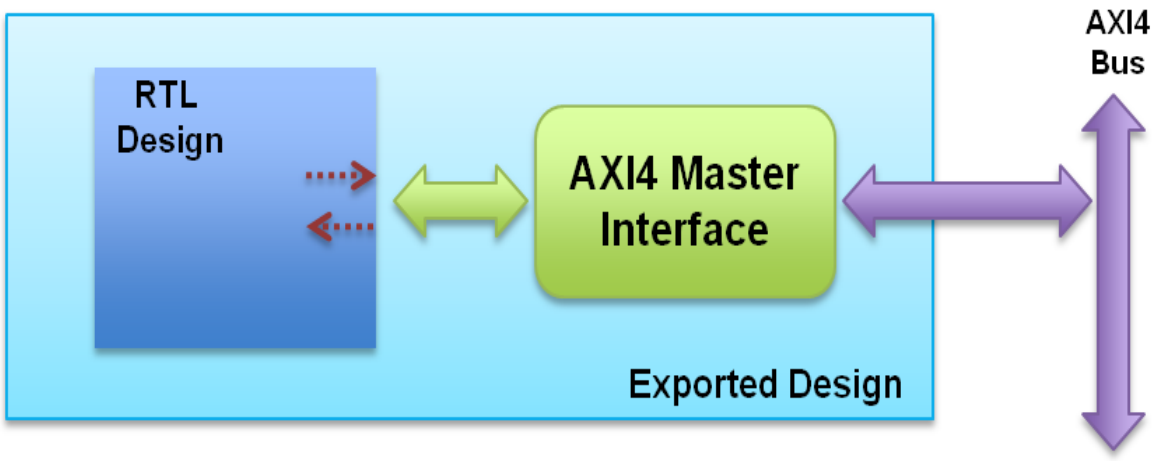

Figure 2-48: AXI4 Master Interface

AXI4 Stream Interface

An AXI4 Stream interface is an AXI4-Stream master/slave interface. This interface can be applied to any ap_fifo RTL port.

When applying a stream interface it is typical to define a struct whose members correspond to the desired AXI4 Stream bus signals. In this example, struct DATA is defined and contains

two members, data and strb. This example requires that both the port map and grouping options are used. If there is only a single data variable, no other options are required.

The AXI4 Stream interface provided by Vivado HLS is designed to fit the flexibility of the bus standard. Most signals (e.g. TREADY, TKEEP, TSTRB) in the AXI4 Stream standard are optional. Only two signals TVALID and TREADY are mandatory (TREADY is optional in the bus standard, but it is used in the Vivado HLS AXI4 Stream interface).

This example maps struct member data to the TDATA signal and struct member strb to the TSTRB signal.

```
port_map={{data_i_data TDATA} {data_i_strb TSTRB}}
port_map={{data_o_data TDATA} {data_o_strb TSTRB}}
```

Using a struct allows multiple variables to be mapped into a single AXI4 Stream. However, the default behavior in Vivado HLS is that each member of a struct creates a separate RTL interface port. The bus bundle option is required to keep all members mapped to the same AXI4 grouped.

Multiple RTL ports can be grouped into a single AXI4 stream interface in the same manner as an AXI4 slave interface. The RTL ports grouped into an AXI4 stream interface however must be either all input ports or all output ports. Bidirectional interfaces are not supported for grouping.

Output interfaces are implemented a AXI4 Stream master interface and input interfaces an AXI4 Stream slave interface.

In this example, the block-level interface protocol is removed by setting the IO protocol to ap_ctrl_none. This ensures there are no other interface signals in this example (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256

typedef struct {
    uint32 data;
    uint4  strb;
} DATA;

void foo_top (DATA data_i[N], DATA data_o[N]) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_fifo port=data_i
#pragma AP interface ap_fifo port=data_o

// Define the pcore interfaces AXI4 slave
#pragma AP resource core=AXI4Stream variable=data_i metadata="-bus_bundle
AXI4Stream_S" port_map={{data_i_data TDATA} {data_i_strb TSTRB}}

// Define the pcore interfaces AXI4 master
```

```
#pragma AP resource core=AXI4Stream variable=data_o metadata="-bus_bundle
AXI4Stream_M" port_map={{data_o_data TDATA} {data_o_strb TSTRB}}

    int i;
    DATA buf[N];

    for (i = 0; i < N; i++) {
        buf[i] = data_i[i];
    }

    for (i = 0; i < N; i++) {
        data_o[i] = buf[N-1-i];
    }

}
```

If the port mappings are applied using the GUI, each port map should be enclosed by braces {RTL_PORT AXI4_INTERFACE_SIGNAL} and all maps enclosed by an outer set of braces. The entire string must be entered into the port_map option box. For example:

```
{{data_o_data TDATA} {data_o_strb TSTRB}}
```

Figure 2-49 shows the interface created from the example shown above, with RTL inputs and outputs grouped into separate interfaces.

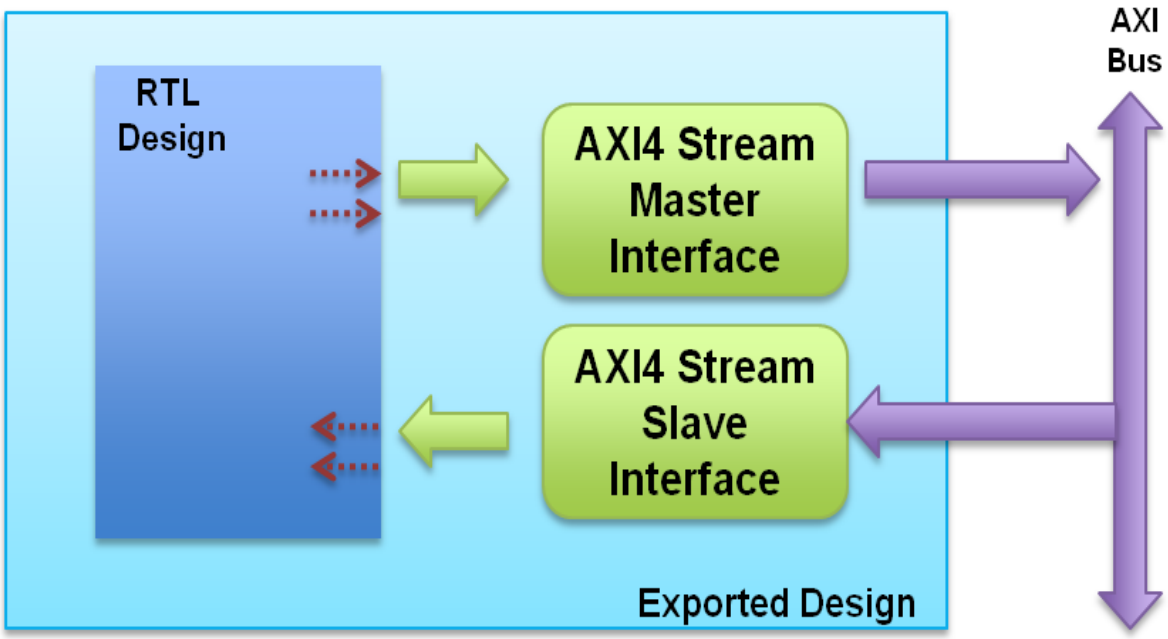

Figure 2-49: Pcore: AXI4 Stream Interfaces

This example used a unique struct. The header file ap_axi_sdata.h, located in the include directory in the Vivado HLS installation area, defines signed and unsigned structs for all

possible signals in the AXI4 interface standard. These can be used in place of creating your own structs.

- ap_axi_s is a signed interpretation of the AXI stream
- ap_axi_u is an unsigned interpretation of the AXI stream

```
template<int D,int U,int TI,int TD>
struct ap_axis{
  ap_int<D>    data;
  ap_uint<D/8> keep;
  ap_uint<D/8> strb;
  ap_uint<U>   user;
  ap_uint<1>   last;
  ap_uint<TI>  id;
  ap_uint<TD>  dest;
};

template<int D,int U,int TI,int TD>
  struct ap_axiu{
    ap_uint<D>   data;
    ap_uint<D/8> keep;
    ap_uint<D/8> strb;
    ap_uint<U>   user;
    ap_uint<1>   last;
    ap_uint<TI>  id;
    ap_uint<TD>  dest;
  };
```

PLB Slave Interface

The following example shows how multiple RTL ports are bundled into a common PLB slave interface. This allows multiple RTL ports to be accessed through a single bus interface.

```
int foo_top (int *a, int *b, int *c, int *d) {

// Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_hs port=b
#pragma AP interface ap_hs port=c
#pragma AP interface ap_hs port=d
#pragma AP interface ap_ctrl_hs port=return register

// Define the pcore interfaces and group into PLB slave "slv0"
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=a
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=b

// Define the pcore interfaces and group into PLB slave "slv1"
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=c
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=d
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=return

    *a += *b;
    return (*c + *d);
}
```

In the example above, ports “a” and “b” are grouped into a common PLB slave interface as shown in Figure 2-50.

The block-level IO protocol is explicitly set by applying interface mode ap_ctrl_hs to the function return. (This is the default but shown explicitly in this example). All block-level IO interface ports (ap_start, ap_done, etc) are assigned to a PLB slave interface as a single group along with the function output (the ap_return port).

Ports “c” and “d” and grouped with the block-level IO protocol signals and function return into group “slv1” as shown in Figure 2-50.

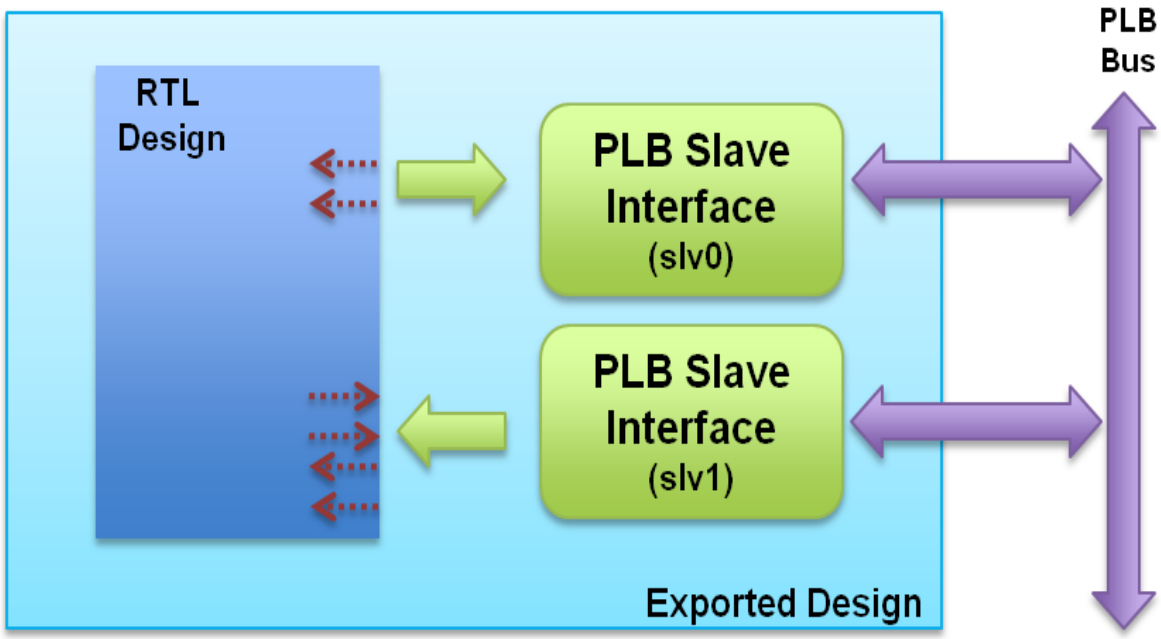

Figure 2-50: Pcore: PLB Slave Interfaces with Bundle Ports

If the RESOURCE directive is applied using the GUI, the entire bus bundle string (e.g. "-bus_bundle slv1") including the quotes should be entered into the metadata option box.

When exported as a Pcore, PLB slave interfaces are created with a set of accompanying C files for use with the processor which will access the slave interface. These C files are similar to those provided with the AXI4 Lite Slave interface and an explanation of the C files is provided in the AXI4 Lite Slave section above.

PLB Master Interface

To create a PLB master interface, the RTL ports must be an `ap_bus` interface, as shown in Table 2-7. This example sets the port “m” as `an ap_bus` and then specifies that the port resource is a “PLB46M” resource.

The block-level interface protocol is removed in this example by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256

typedef uint32 DT;
void foo_top (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as a PLB master
#pragma AP resource core=PLB46M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

This specification results in a PLB v4.6 master interface which can be connected to a PLB bus and will act as a master interface.

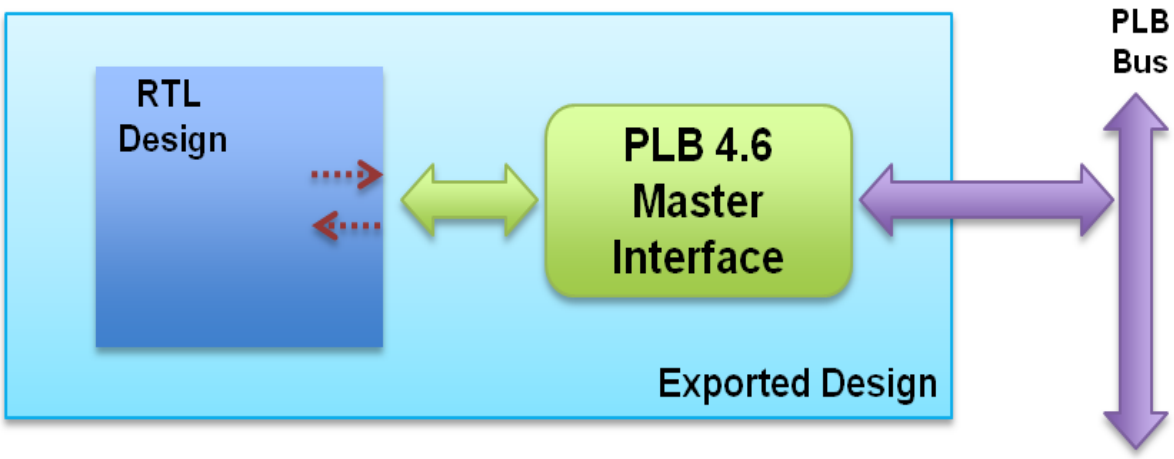

Figure 2-51: Pcore: PLB Master Interface

NPI Interface

An NPI interface is created by first creating an ap_bus on the RTL interface. This interface can then be assigned a "NPI64M" resource as shown in the following example.

```
#include "ap_cint.h"

#define N 256
typedef uint32 DT;

void foo_top (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as an NPI master
#pragma AP resource core=NPI64M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

In this example, the block-level interface protocol is removed by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

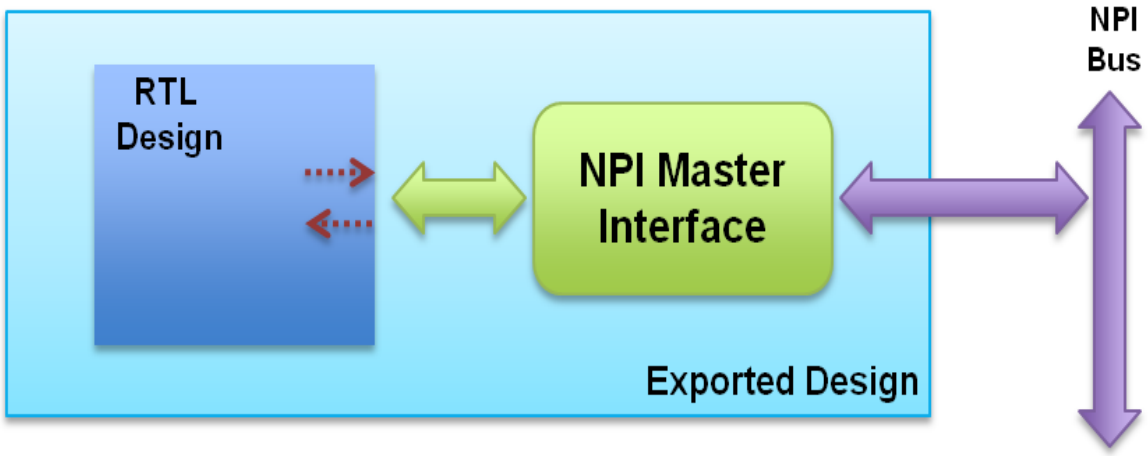

Figure 2-52: Pcore: NPI Master Interface

FSL Interface

An FSL interface can be connected to an RTL ap_fifo interface. The FSL interface is a master/slave interface.

- If the interface is an input, a FSL slave interface is generated.
- If it is an output, a FSL master interface is generated.
- FSL interfaces cannot be connected to bi-directional ports (an `ap_fifo` interfaces cannot be applied to these ports).

```
#define N 256

void foo_top (int data_i[N], int data_o[N]) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_fifo port=data_i
#pragma AP interface ap_fifo port=data_o

// Define the pcore interfaces as FSL types
#pragma AP resource core=FSL variable=data_i
#pragma AP resource core=FSL variable=data_o

    int buff[N], i;
    for (i = 0; i < N; i++) {
        buff[i] = data_i[i];
    }
    for (i = 0; i < N; i++) {
        data_o[i] = buff[N-1-i];
    }
}
```

In this example, there are two `ap_fifo` interfaces. One is an input and the other an output. This examples set the block-level protocol to `ap_ctrl_none` to ensure there are no other ports (there is no function return and the `ap_ctrl_none` ensures there are no block-level handshake signals).

The input and output FSL interfaces are shown in Figure 2-53.

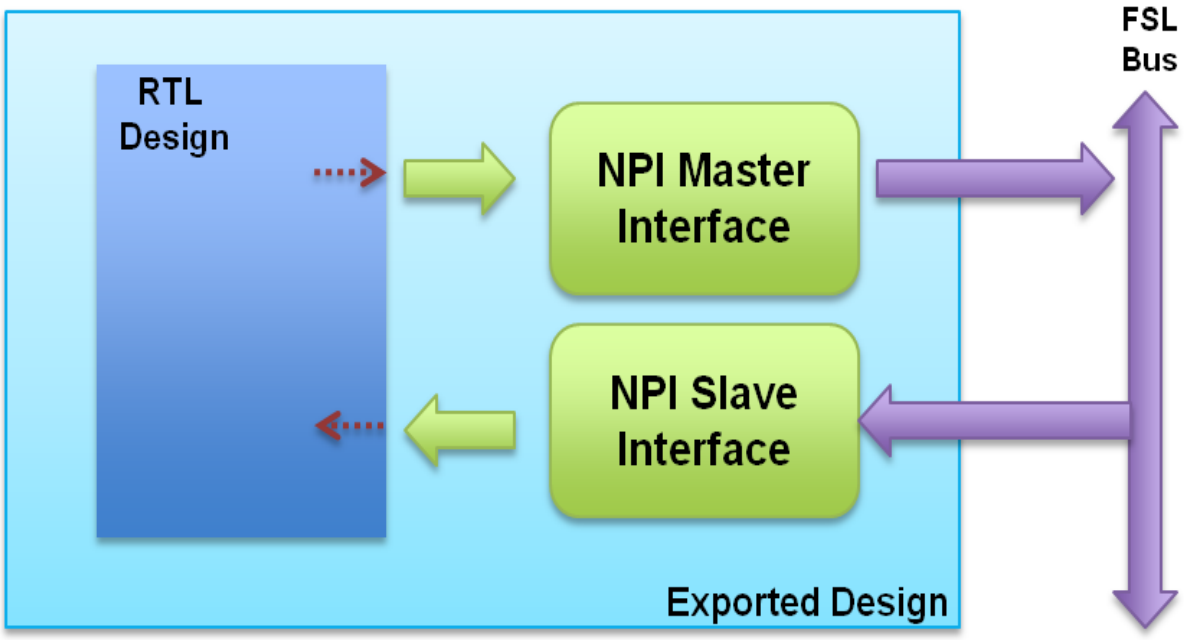

Figure 2-53: Pcore: FSL Master & Slave Interface

SystemC Interface Synthesis

In general, interface Synthesis is not supported for SystemC designs. The IO ports for SystemC designs should be fully described in the SC_MODULE interface and in behavior of the ports fully described in the source code.

An exception to this is memory ports. Given a design with standard SystemC array ports:

```
SC_MODULE(my_design) {
  //"RAM" Port
  sc_uint<20>  my_array[256];
  …
```

The port "my_array" would be synthesized into internal RAMs.

Including the High-Level Synthesis header file "ap_mem_if.h" allows the port to be specified as an ap_mem_port<data_width, address_bits> port. An ap_mem_port will be synthesized into a standard RAM interface port with the specified data and address bus-widths.

The code example above, can be transformed into the following:

```
#include "ap_mem_if.h"

SC_MODULE(my_design) {
  //"RAM" Port
  ap_mem_port<sc_uint<20>,sc_uint<8>, 256> my_array;
```

```
…
```

This will ensure interface port “my_array” is implemented as a RAM interface.

When an ap_mem_port is added to a SystemC design, an associated ap_mem_chn must be added to the SystemC test bench to drive the ap_mem_port.

In the test bench, an ap_mem_chn is defined and attached to the instance as shown:

```
#include "ap_mem_if.h"
…
ap_mem_chn<int,int, 68> bus_mem;
…

// Instantiate the top-level module
my_design U_dut (“U_dut”)
U_dut.my_array.bind(bus_mem);
…
```

The header file “ap_mem_if.h” is located at “$VIVADO_HLS_ROOT\include \ap_sysc\ap_mem_if.h” and must be included during SystemC simulation.

Manual Interface Specification

High-Level Synthesis has the ability to identify blocks of code which defines a specific IO protocol. This allows an IO protocol to be specified without using interface synthesis or SystemC (the protocol directive explained below can also be used with SystemC designs to provide greater IO control).

In following example code:

- input "response[0]" is read
- output "request" is written
- input "response[1]" is read.
- AND it is required that the final design perform the IO accesses in this order

```
void test (
  int   *z1,
  int   a,
  int   b,
  int   *mode,
  volatile int  *request,
  volatile int  response[2],
  int   *z2
  ) {

  int   read1, read2;
  int   opcode;
  int   i;


  P1: {
```

```
    read1       = response[0];
    opcode      = 5;
    *request    = opcode;
    read2       = response[1];
  }
  C1: {
          *z1   = a + b;
          *z2   = read1 + read2;
  }
}
```

When High-Level Synthesis implements this code there is no reason the "request" write should be between the two reads on "response". The code is written with this IO behavior but there are no dependencies in the code which enforce it. High-Level Synthesis may schedule the IO accesses in the same manner or choose some other access pattern.

In this case the use of a protocol block can be used to enforce a specific IO protocol behavior. Since the accesses occur in the scope defined by block "P1", a protocol can be applied to this block as follows:

- Include "ap_utils.h" header file which defines ap_wait().
- Place an ap_wait() statement after the write to "request", but before the read on "response[1]".
  - The ap_wait() statement will not cause the simulation to behave any differently, but it will instruct High-Level Synthesis to insert a clock here during synthesis.
- Specify that block P1 is a protocol region.
  - This will instruct High-Level Synthesis that the code within this region is to be scheduled as is: no re-ordering of the IO or ap_wait() statements.

Applying the directive as shown:

```
set_directive_protocol test P1 -mode floating
```

To the modified code:

```
#include "ap_utils.h"// Added include file

void test (
  int   *z1,
  int   a,
  int   b,
  int   *mode,
  volatile int  *request,
  volatile int  response[2],
  int   *z2
  ) {

  int   read1, read2;
  int   opcode;
  int   i;
```

```
  P1: {
    read1       = response[0];
    opcode      = 5;
    ap_wait();// Added ap_wait statement
    *request    = opcode;
    read2       = response[1];
  }
  C1: {
          *z1   = a + b;
          *z2   = read1 + read2;
  }
}
```

Will result in exact IO behavior specified in the code:

- input "response[0]" is read
- output "request" is written
- input "response[1]" is read.

The -mode floating option allows other code to execute in parallel with this block, if allowed by data dependencies. The fixed mode would prevent this.

Design Optimization

This chapter outlines the various optimizations and techniques which can be employed to direct High-Level Synthesis to produce a micro-architecture which satisfies the desired performance, area and power goals.

The layout of the topics chapter is designed to help minimize the effort required in finding the correct optimizations to apply and maximize productivity. It is recommended to approach design optimization by reviewing the topics in the following order:

- Checklist & Guidelines
- Clocks, Timing & RTL output
- Arbitrary Precision Data Types
- Performing Optimizations

The "Checklist and Guidelines" section provides strategies for applying the optimizations discussed in this chapter. Optimization strategies are provided for a number of standard design goals such as area, throughput, latency and power. However, even if your overall goals are different from one of these typical design goals, a review of this section will provide a solid foundation for applying High-Level Synthesis's features and capabilities to meet your goals.

In high-level synthesis, like logic synthesis, subtle changes to the input source code or the constraints can result in a somewhat different output design, only more so. To prevent the need to "start over again" review the sections on "Clocks, Timing & RTL output" and "Arbitrary Precision Data Types" to ensure the basic design parameters and design description are both correct and as ideal as possible before spending time and effort time applying more complex optimizations.

When the time comes to apply more complex and powerful optimization techniques a top-down or bottom-up approach can be used.

- If the design is far from meeting its latency or throughput goals, work from the top-down. Applying optimizations at higher levels of abstraction and operation is more likely to result in large improvements: start reviewing the design at the function level and work down towards the logic level. This is also the order presented in this User Guide.
- If the design is almost achieving its goals, focus on trying to reduce a few clock cycles or resources. In this case, it may be more productive to start at the level of logic structures, reviewing if better component selection would allow more operations in a

cycle or if array accesses are causing bottlenecks and work up toward the function level.

Checklist & Guidelines

This section outlines some basic strategies for quickly reaching your optimization goals.

Design Basics

Review the basic designs parameters discussed in the section "Clocks, Timing & RTL output" to ensure the RTL is being created with the following:

- Correct clock and clock uncertainty.
- Reset style.
- State encodings.

Interface Synthesis

It is important to pay attention to both algorithm and interface synthesis (the interface has to provide the data, and could very well be the bottleneck to achieve required throughput rates).

Before proceeding to perform design optimizations, ensure the correct interfaces are being used:

- Confirm the current interfaces on the design are compatible with whatever design they will communicate with.
- Review the "Interface Management" chapter to select and define the correct interface.
- Confirm that the chosen interfaces work with the post-synthesis verification methodology, as changing the interfaces later may result in a different schedule. Refer to the "Verification" chapter for more details on the verification flow.

Finally, it should be appreciated that to meet performance requirements it may be necessary to change how the interface is implemented or the type of interface that is used. For example, an array interface may benefit from:

- Being changed from a single to dual-port RAM implementation
- Being changed from a RAM to streaming FIFO implementation
- Being changed from an array to a pointer which can support a DMA-like bus interface.

These types of decisions and changes are no different from those made when optimizing by hand: sometimes the biggest gain or only way forward is to consider a change at the level above the design and hence change an interface protocol. High-Level Synthesis will quickly allow you to determine if such changes will result in a beneficial architecture.

Data Types and Bit-Widths

High-Level Synthesis will propagate data types based on arithmetic properties; however it is safer to be explicit where possible.

The bit-widths of the variables in the C function directly impact the size of the storage elements and operators used in the RTL implementation. If a variables only requires 8-bits but is specified as an integer type (32-bit) it may result in larger and slower 32-bit operators being used, reducing the number of operations which can be performed in a clock cycle and potentially increasing latency and throughput.

- Use the appropriate precision for the data types. Refer to section “Arbitrary Precision s” in this chapter.
- Confirm the size of any arrays which are to be implemented as RAMs or registers. The area impact of any over-sized elements in the array is magnified because there are multiple elements in each array.
- Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they will negatively impact both area and performance.
- If using ANSI C, use apcc after modifying any bit-widths to confirm existing simulations give the same results. Refer to section “Arbitrary Precision ” below for an explanation of why apcc is required.

Minimum Area Designs

When trying to minimize the area in a design concentrate on re-using functions and loops. Functions and loops will iterate over the same hardware resources each time they execute: this maximizes sharing at a level above the operator.

- Before beginning work on minimizing the area, ensure the design already meets, or is close to meeting, its performance metrics.
- If a function is called multiple times, it will share the same hardware. This is a great way to save area through coding style. Check to see if function inlining will allow more functions to be shared (Refer to the section on “Function Re-use, Inlining”).
- Similarly, a loop will iterate over the same hardware. This implements the loop functionality with a small area but a large latency. Ensure for-loops are not unrolled: the default is to keep them rolled. (Refer to the section "Loop Optimizations").
- If the design is pipelined, see if a different initiation interval still satisfies the throughput requirements but allows greater re-use of components. (Refer to the section on “Function Optimizations” and "Loop Optimizations").
- Check if arrays are being optimally implemented on the existing RAMs or if array partitioning or reshaping could make more optimal use of the available RAM resources: allowing more parallel accesses. (Refer to "Array Optimizations").

Maximum Throughput Designs

In a maximum throughput design, the challenge is to process as much data as possible in as few cycles as possible.

- Starting at the function level, and especially in C and C++ designs (SystemC can support concurrency natively) examine the section on “Function Dataflow Pipelining”.
- Review the "Loop Pipelining" section later in this chapter to determine if pipelining can be applied on loops.

If all pipelining is complete or cannot be considered, the next step is to look at minimizing the latency through each function and loop: if it requires less cycles to complete the operations, the next input can be read sooner. The techniques for minimizing the latency are given in the next section but the following are areas of importance when dealing with design throughput.

- Review the summary section of the reports. Examine the report on each function in the hierarchy for critical loops, those having the greatest impact, and use some of the techniques in the “Minimum Latency Designs” section below to reduce the latency of critical loops.
  - A loop which takes 25 clocks but is executed twice has a 50 cycle impact.
  - A loop which requires only 4 clocks but is executed 256 times has a 1024 cycle impact and should be considered more “critical”.
- Finally, if the limitation is at the design ports, consider changing the type of interface. This will require confirming with the other designs in the system that such changes can be supported.

Minimum Latency Designs

To reduce the latency through a design:

- Starting at the function and loop levels, and especially in C and C++ designs (SystemC can support concurrency natively) examine the sections on “Function Dataflow Pipelining” and "Loop Dataflow Pipelining" to improve concurrency.
- If there are any for-loops, check to see if unrolling or partially unrolling them, as described in section "Unrolling Loops", will reduce the latency. Unrolling allows more operations to be done in parallel using less cycles (but more resources and larger area).
- If there are multiple loops, remember that it costs 1 clock cycle to move from one loop-body to another. Review the sections on “Merging Loops” and “Flattening Nested Loops” to reduce loop overheads.
- Be careful with arrays. They typically map onto memories, which have a limited access capabilities (read ports and write ports). This can result in dependencies in the hardware that can increase the latency. For example, a dual-port RAM, or reconfigured

RAM as discussed in "Array Optimizations", may allow more reads and writes in the same clock cycle.

Minimum Power Designs

High-Level Synthesis automatically performs a number of power optimizations. Examples of these include operator gating, which is performed during scheduling and the addition of pipeline enables added to the beginning of pipelines. All power optimizations are applied automatically, if they do not negatively impact performance constraints.

High-Level Synthesis does not allow performance to be sacrificed to improve power however there are a number of methodologies which can be followed to further reduce power.

- Review the section above on "Minimum Area Designs" as reducing area can have a great impact on power.
- Do not forget to use the architectural exploration capabilities of High-Level Synthesis. High-Level Synthesis allows you change the clock period and quickly re-generate a new micro-architecture.

Clocks, Timing & RTL output

For C and C++ designs only a single clock is supported. The same clock is applied to all functions in the design. For SystemC designs, each SC_MODULE may be specified with a different clock.

To specify multiple clocks in a SystemC design, use the -name option of the create_clock command to create named clocks and use the set_directive_clock command or pragma to specify which function contains the SC_MODULE to be synthesized with the specified clock. Each SC_MODULE can only be synthesized using a single clock: clocks may be distributed through functions, such as when multiple clocks are connected from the top-level ports to individual blocks, but each SC_MODULE can only be sensitive to a single clock).

The clock period, in ns, is set in the solutions setting of the GUI. In addition High-Level Synthesis uses the concept of a clock uncertainty to provide a user defined timing margin.

High-Level Synthesis estimates the timing of operations in the design but it cannot know the final component placement and net routing: as such, it cannot know the exact delays. The clock uncertainty is a value which is subtracted from the clock period to give the usable clock period as shown in Figure 2-54. High-Level Synthesis will use the usable clock period to schedule the operations in the design.

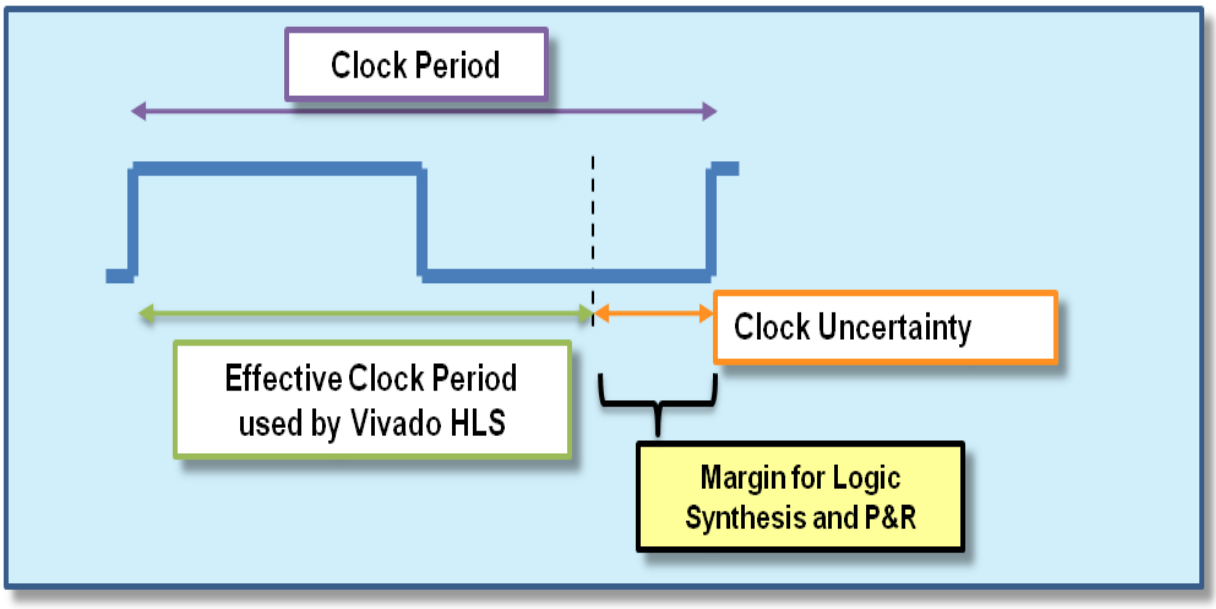

Figure 2-54: Clock Period and Margin

This provides some user specified slack to ensure downstream processes, such as logic synthesis and place & route, have enough timing margin to complete their operations and limits the design iterations associated with timing closer.

By default, the clock uncertainty is 12.5% of the cycle time; however the clock uncertainty can be defined by the user, in the solutions settings menu beside the clock period. The advantage of the automatic setting is that it is updated if the clock period is changed (else the user must manually update the clock uncertainty when the clock period is changed).

Timing

The timing information used for the RTL operators and registers is defined by the library. The libraries are all pre-characterized and stored within High-Level Synthesis.

High-Level Synthesis will always aim to meet latency, throughput (initiation interval) and the timing constraints. However, even when High-Level Synthesis cannot meet constraints, it will always output an RTL design.

- If High-Level Synthesis cannot meet a throughput constraint due to a data dependency (for example, if a throughput of one is required but it requires two cycles to read a value from memory) it will automatically increase the throughput until a design can be realized.
- If the timing constraints inferred by the clock period cannot be met High-Level Synthesis will issue message SCHED-644, as shown below, and output a design with the best achievable performance.

```
@W [SCHED-644] Max operation delay (<operation_name> 2.39ns) exceeds the effective
cycle time
```

Even if High-Level Synthesis cannot meet timing requirements for a particular path, or paths, it will still endeavor to achieve timing on all other paths. This behavior allows the user to evaluate if higher optimization levels or special handling of those failing paths by downstream processes can pull-in and ultimately satisfy the timing.

IMPORTANT: *It is important to review the constraint report after synthesis to determine if all constraints have been met: the fact that High-Level Synthesis produces an output design does not guarantee the design meets all performance constraints. Review the "Performance Estimates" section of the design report.*

A design report is generated automatically for each function in the hierarchy when synthesis completes and can be viewed in the solution reports folder and is stored in the solution directory in file ./syn/report/<function name>.rpt.

The worse case timing for the entire design is reported as the worst case in each function report. (There is no need to review every report in the hierarchy). If the timing violations are too severe to be further optimized and corrected by downstream processes, review the techniques in the remainder of this chapter before considering a faster target technology.

RTL Output

Various characteristics of the RTL output by High-Level Synthesis can be controlled using the config_rtl command. These include

- The ability to specify the type of FSM encoding used in the RTL state machines.
- The ability to add an arbitrary comment string, such as a copyright notice, to all RTL files using the –header option.
- Using the –prefix option to add a unique prefix to all output files, allowing RTL files generated from the same top-level function (and which would have the same name by default) to be easily combined in the same directory.

The RTL configuration can be defined via the Solution menu, using Solution > Solution Settings > General tab > config_rtl as shown in Figure 2-55.

Using the Menu

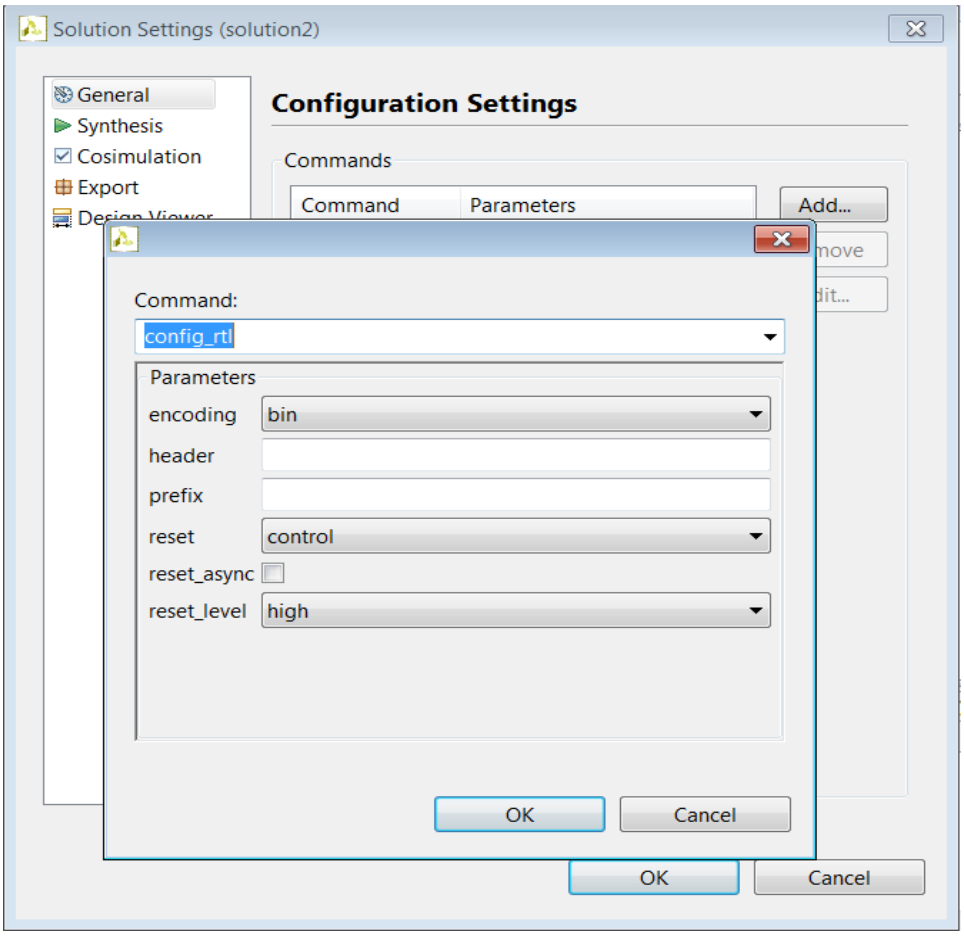

Figure 2-55: RTL Configurations

Typically the most important aspect of RTL configuration is selecting the reset behavior.

Selecting RTL Reset Behavior

When discussing reset behavior it is important to understand the difference between initialization and reset.

In C, variables defined with the static qualifier and those defined in the global scope, are by default initialized to zero. Optionally, these variables may be assigned a specific initial value. For these type of variables, the initial value in the C code is assigned at compile time (at time zero) and never again. In both cases, the same initial value will be implemented in the RTL after synthesis: this ensures the FPGA bit-stream will initialize the device with the same initial value(s).

Any variable not defined with the static qualifier or defined in the global scope, but given an initial value, is assigned its initial value each time the function executes. As such, the initialization of such variables will occur each time the RTL design starts and will be part of

normal operation. An example of such an initialization is the accumulator in an FIR filter. When a new data samples arrives, the accumulator is set to zero, values are then calculated and accumulated but the result of the accumulation is not carried forward to the next transaction. When a new sample arrives the accumulator is once again set to zero.

In addition to replicating the initial value of static and global variables in the RTL, a reset can optionally be applied to the RTL (the current default is to add a reset).

Note: Note: Reset is separate and in addition to initialization.

Some configurations of the reset do not re-initialize variables to their initial power-up state when the reset signal is applied. In these cases, the initial value is only applied at power-up time.

The current default operation is not to re-initialize static and global variables when a reset occurs.

The behavior of the RTL reset is controlled using the RTL configuration settings shown in Figure 2-55.

This includes being able to set the polarity of the reset and whether the reset is synchronous or asynchronous (Xilinx technologies favor the use of a synchronous reset: the default) but more importantly it determines, via the reset option, which registers are reset when the reset signal is applied.

The reset option has four settings:

- **none**: no reset is added to the design.
- **control**: reset control registers, such as those used in state machines and to generate IO protocol signals.
- **state**: reset control registers and registers/memories derived from static/global variables in the C code. Any static/global variable initialized in the C code is reset to its initialized value.
- **all**: reset all registers and memories in the design. Any static/global variable initialized in the C code is reset to its initialized value.

The default is setting control.

Note: When option state is used, any arrays implemented with RAMs will typically be initialized after reset. Remember, most arrays implemented as a RAM are defined as statics and therefore imply that all elements be initialized to zero: even if the elements do not explicitly initialization.

For a large memory, this reset behavior may take many clock cycles and required more area resources to implement.

Example Tcl commands specifying all the attributes discussed in this section are shown below (additional options can be reviewed in the config_rtl man page). These commands perform the following:

- Set clock period as 10 ns

- Add a technology library
- Set the clock uncertainty to 2ns (if not specified, this defaults to 12.5% of the clock period).
- Create an RTL output with active low asynchronous reset on all registers.
- Use one-hot encoding for all state machines.

```
open_solution solution2

set_part  {xc6vlx365tff1759-3}

create_clock -period 10ns
set_clock_uncertainty 2

config_rtl my_func -encoding onehot –reset all -reset_level low -reset_async
```

Once these basic specifications are set and defined, the design can be optimized as outlined in the remaining sections of this chapter.

Function Optimizations

A complete list of the optimizations which can be specified upon a function can be seen in the GUI (Figure 2-56):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select a function, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

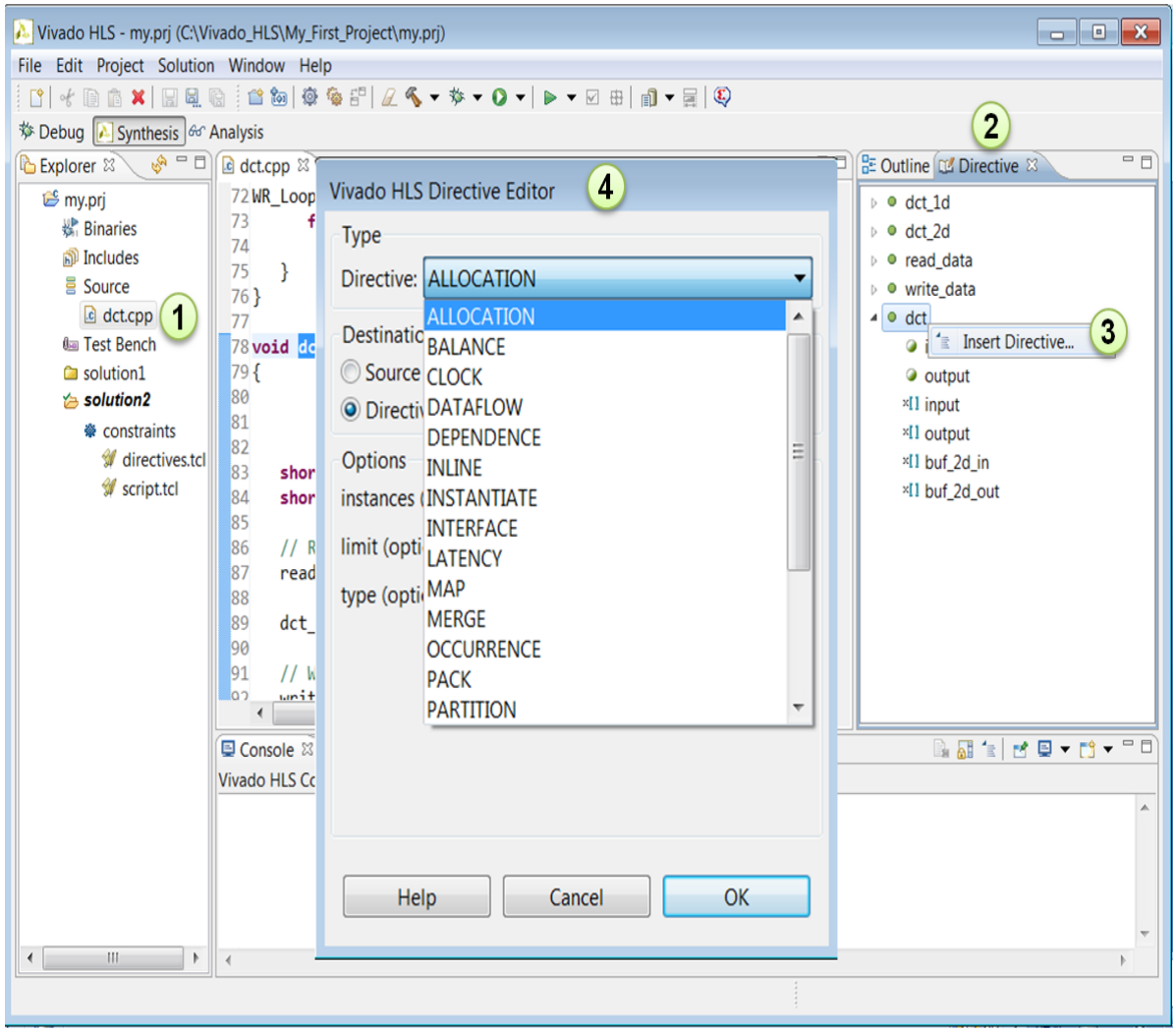

Figure 2-56: GUI Function Directives

TIP: *Not all of the directives which can be applied at the function level are related to the optimizations of functions.*

For example, when applied to a function, the RESHAPE directive is applied to all arrays in the function (in effect, a short-hand way to apply the directive to multiple arrays simultaneously). Array optimizations are discussed in a subsequent chapter.

Table 2-8 lists the directives which impact the behavior and optimization of functions themselves and the order in which the optimizations are discussed in this chapter.

Table 2-8: Function Optimizations

GUI Directive	Description
Inline	Inlines a function, removing all function hierarchy. Used to help improve latency and throughput by reducing function call overhead.
Instantiate	Allows functions to be locally optimized.
Dataflow	Enables concurrency at the function level and used to improve throughput and latency.
Pipeline	Improves throughput of the function by allowing the concurrent execution of operations within a function.
Latency	Allows a minimum and maximum latency constraint to be specified on the function.
Interface	Applies function level handshaking. The synthesized control ports start, done and idle enables test bench re-use. This option is more fully discussed in the "Interface Management" chapter.

Function Re-use, Inlining & Instantiation

High-Level Synthesis supports the synthesis of a hierarchy of function calls. By default, each function gets mapped to a specific hardware implementation. Calling the same function multiple times within the same enclosing function reuses the same hardware, just like instantiating a block in an RTL design.

Function re-use is a highly effective way of ensuring resource sharing and keeping the design smaller, since it guarantees all operations within the function are shared. If the function hierarchy is removed, it is unlikely the same level of sharing can be achieved by trying to re-assemble the individual operations at multiple locations (it's more likely that local optimizations at different locations will limit sharing).

Function Inlining

There is typically a cycle overhead to enter and exit functions and removing the function hierarchy can mean improved latency and throughput.

Function inlining can be used to remove function hierarchy, often at the expense of area. If however a function is only called a few times and/or is small, inlining the function may

actually improve area by allowing the few components within the function to be better shared. There are other situations where inlining a function provides benefits.

- Improve function sharing
- Allow architecture exploration

All instances of a function will have the same implementation but to ensure functions are shared they must be called within the same enclosing function and at the same level of hierarchy. This may require inlining some other functions.

In this code example, function calls "foo_1" and "foo_2" may be shared but not function call "foo_3" which is at a different level of hierarchy.

```
foo {x,y} {
  ...
}
foo_sub (p, q) {
  int q1 = q + 10;
  foo(p1,q);// foo_3
  ...
}
void foo_top { a, b, c, d} {
  ...
  foo(a,b);//foo_1
  foo(a,c);//foo_2
  foo_sub(a,d);
  ...
}
```

In the above example, function inlining can be used to increase sharing by removing the hierarchy created by function "foo_sub". This can be performed in the GUI as shown in Figure 2-56 or using the set_directive_inline command.

```
set_directive_inline foo_sub
```

The inlining directive optionally allows all functions below the specified function to be recursively inlined: if the -recursive option was used in the above example, function "foo_sub" would be inlined as would function call "foo_3". If the recursive option is used on the top-level function, all function hierarchy in the design will be removed.

The -off option can optionally be applied to functions to prevent them being inlined. If the following commands are applied to the example above:

```
set_directive_inline –region -recursive foo_top
set_directive_inline -off foo_sub
```

All functions in the region of "foo_top" would be inlined, recursively down the hierarchy. Function "foo" would be inlined for function calls "foo_1" and "foo_2" but not "foo_3", which is inside "foo_sub" since the -off option applied on function "foo_sub" will prevent inlining.

Function inlining is a powerful way to substantially modify the structure of the code without actually performing any modifications to the source and provides a very powerful method for architectural exploration.

Function Instantiation

Function instantiation is a an optimization technique which has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

Function instantiation exploits the fact that some inputs to a function may be a constant value when the function is called and uses this to both simplify the surrounding control structures (which are typically creating the constants) and produce smaller more optimized function blocks. This is best explained by example.

Given the following code:

```
void A(){
  B(true);
  B(false);
  B(true);
  B(false);
}

void B(bool mode){
  if (mode) {
     // code segment 1
  } else {
     // code segment 2
  }
}
```

It is clear that function "B" has been written to perform multiple but exclusive operations (depending on whether mode is true or not). Each instance of function "B" will be implemented in an identical manner: this is great for function re-use and area optimization but means that the control logic inside the function must be more complex.

Function instantiation can be performed from the directives tab in the GUI, by inserting pragmas into the code or with the following Tcl command:

```
set_directive_function_instantiate -variable mode=true B
```

After function instantiation, the code will effectively be transformed to have two separate functions, each optimized for different possible values of mode, as shown:

```
void A(){
  B1();
  B2();
  B1();
  B2();
}
```

```
void B1() {
  // code segment 1
}

void B2() {
  // code segment 2
}
```

Each version of the new function "B", that is functions "B1" and "B2", have simplified control structures.

If the function were called at different levels of hierarchy such that function sharing is difficult without extensive inlining or code modifications, function instantiation can provide the best means of improving area: many small locally optimized copies are better than many large copies which cannot be shared.

Function Dataflow Pipelining

Dataflow pipelining takes a sequential functional description (Figure 2-57) and creates a parallel process architecture from it (Figure 2-58). Dataflow pipelining is a very powerful method for improving design throughput.

Figure 2-57: Sequential Functional Description

Figure 2-58: Parallel Process Architecture

The channels shown in Figure 2-58 ensure a function is not required to wait until the previous function has finished all operations before it can begin. Figure 2-59 shows how dataflow pipelining allows the execution of functions to overlap, increasing the overall throughput of the design and reducing latency.

In Figure 2-59(A) the implementation with dataflow pipelining requires 8 cycles before a new input can be processed by "func_A" and 8 cycles before an output is written by "func_C" (assume the output is written at the end of "func_C").

In Figure 2-59(B), "func_A" can begin processing a new input every 3 clock cycles (increased throughput) and it only requires 5 clocks to output a final value (shorter latency).

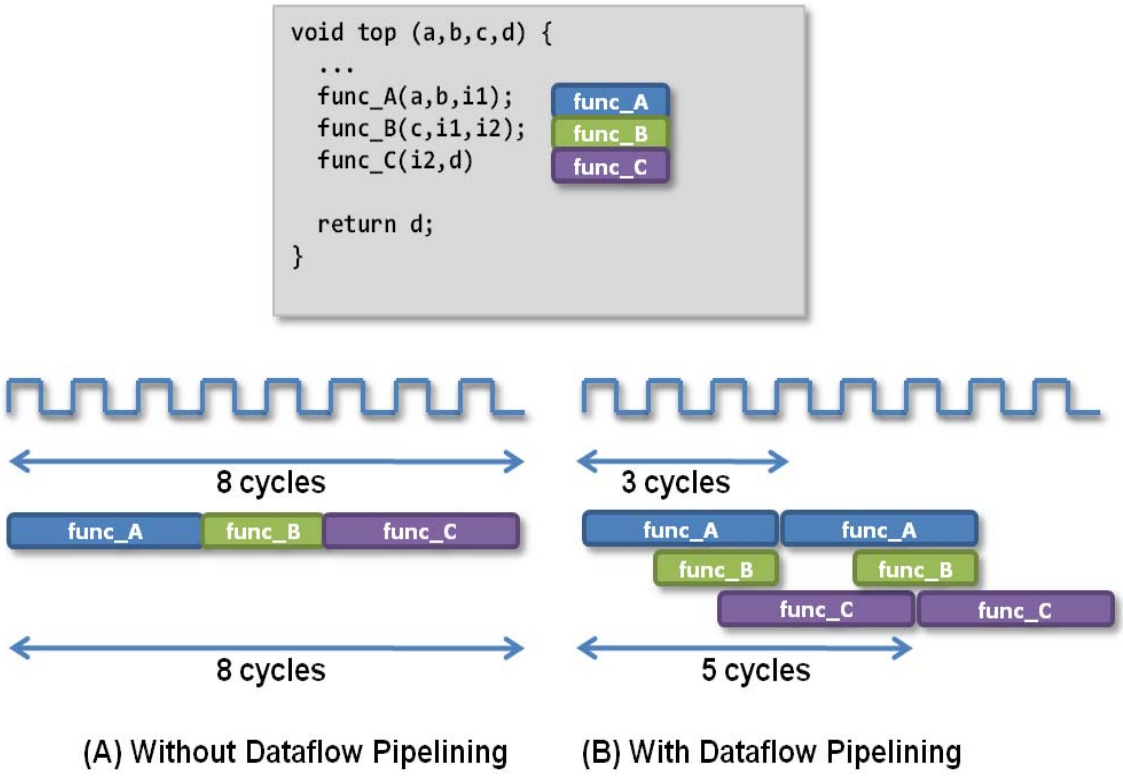

Figure 2-59: Dataflow Pipelining Behavior

The channels between the processes are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data.

- If the function parameter (producer or consumer) is an array the channel is implemented as a ping-pong buffer using standard memory accesses (with associated address and control signals).
- For scalar, pointer and reference parameters and the function return, the channel is implemented as a FIFO, which uses less hardware resources (no address generation) but requires that the data is accessed sequentially.

IMPORTANT: *To use dataflow pipelining the arguments in each function must appear only twice: once as a producer from one function call (including return arguments) and once as a consumer in another function argument.*

In addition to using the GUI directive tab, dataflow pipelining can be specified using the set_directive_dataflow command. When the directive is specified on a function, High-Level Synthesis will seek to improve the concurrency of the functions within it. For the example

shown in Figure 2-59 the following command would perform function dataflow pipelining on functions "func_A", "func_B" and "func_C".

```
set_directive_dataflow top
```

When dataflow pipelining is used, High-Level Synthesis will by default attempt to execute each RTL block starting at the same clock edge: maximum parallel behavior. If data dependencies prevent the RTL implementation of a function from executing until an earlier function provides data (as in the Figure 2-59(B) where "func_B" must wait 1 clock cycle for "func_A" to generate the data for "i1") High-Level Synthesis will automatically adjust the interval between one block starting execution and the next block starting execution, to the minimum possible number of cycles.

The –interval option can be used to specify exactly how many cycles there will be between an RTL block beginning execution and the next RTL block beginning execution. For example, if an interval of 3 is specified, there would be 3 cycles between the start of each function in Figure 2-59(B).

For scalar values, the maximum channel size will be one: only one value is passed from one function to another. When arrays are used as function arguments the number of elements in the channel (memory) is defined by the maximum size of the consumer or producer array. High-Level Synthesis does however provide a means to specify a default channel depth (refer to "Configuring " below).

When dataflow pipelining is applied to a function only the sub-functions at the current level of hierarchy will be pipelined. If a sub-function itself contains additional functions which could also benefit from dataflow pipelining, the sub-function should be inlined to ensure all functions are at the same level of hierarchy.

Configuring Dataflow Memories

The default channel used between function interfaces can be specified using the config_dataflow command. Configuration commands allow a default operation to be set for a solution. This command allows the default channel size and implementation to be set for all channels in a design.

```
config_dataflow -default_channel (fifo | *pingpong*) -fifo_depth <FIFO size>
```

The size of a channel is defined by the maximum size of the consumer or producer array. In some cases this may be overly conservative. The -fifo_depth option provides a means for the user to override the default behavior.

IMPORTANT: *If an array parameter is specified to use a FIFO channel, the array must be set to a streaming type array (refer to "Array Streaming" for an explanation of streaming).*

If the default channel type is FIFO but a specific array has been specified as non-streaming using set_directive_array_stream command, the channel implementation for that array will default to a ping-pong channel (an explicit directive overrides a configuration).

Function Pipelining

Where dataflow pipelining allows the optimization of the communication between functions to improve throughput, function pipelining optimizes the operations within a function and has a similarly positive effect on throughput.

The throughput improvements in function pipelining are shown in Figure 2-60. Function pipelining allows operations to happen concurrently: the function does not have to complete all operations before it begin the next operation.

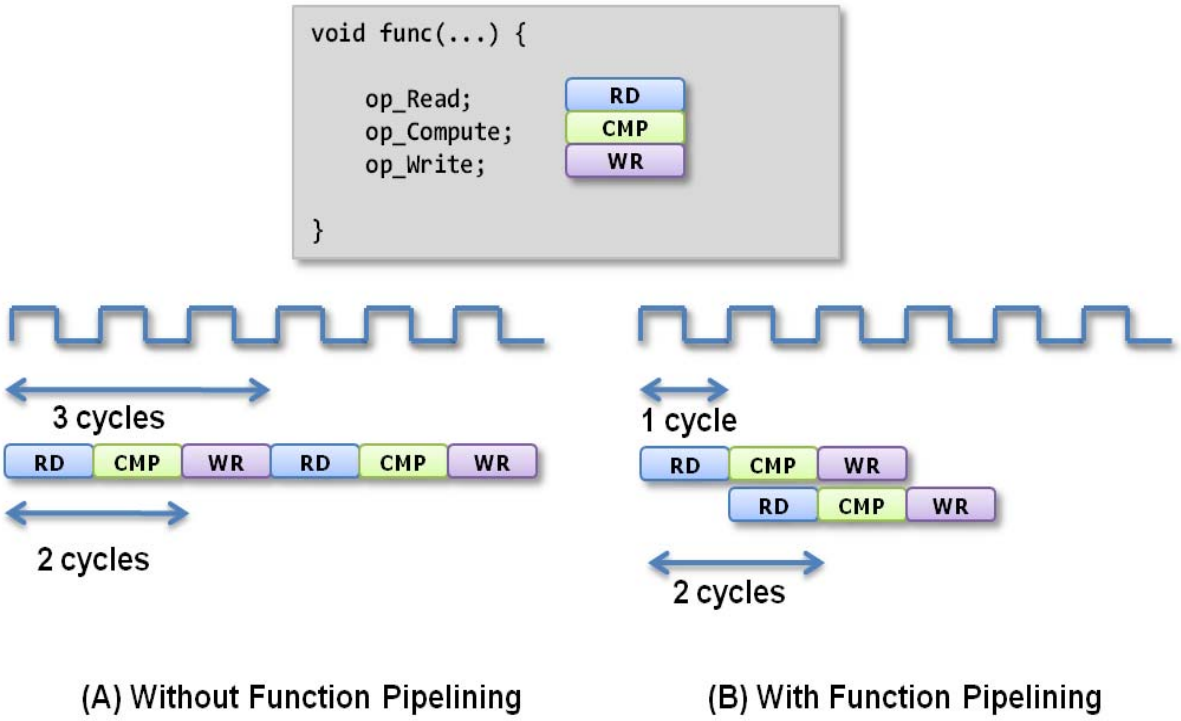

Figure 2-60: Function Pipelining Behavior

Without pipelining the function reads an input every 3 clock cycles and outputs a value every 2 clock cycles. With pipelining, a new input is read every cycle with no change to the output latency or resources used.

Function pipelining is only possible so long as there is no resource contention or data dependency which prevents pipelining. For example, in Figure 2-61 below, assume the input array "m[2]" is implemented with a single-port RAM. The function cannot be pipelined, as shown in Figure 2-61(A) because the two reads operations on input "m[2]" ("op_Read_m[0]" and "op_Read_m[1]") cannot be performed in the same clock cycle.

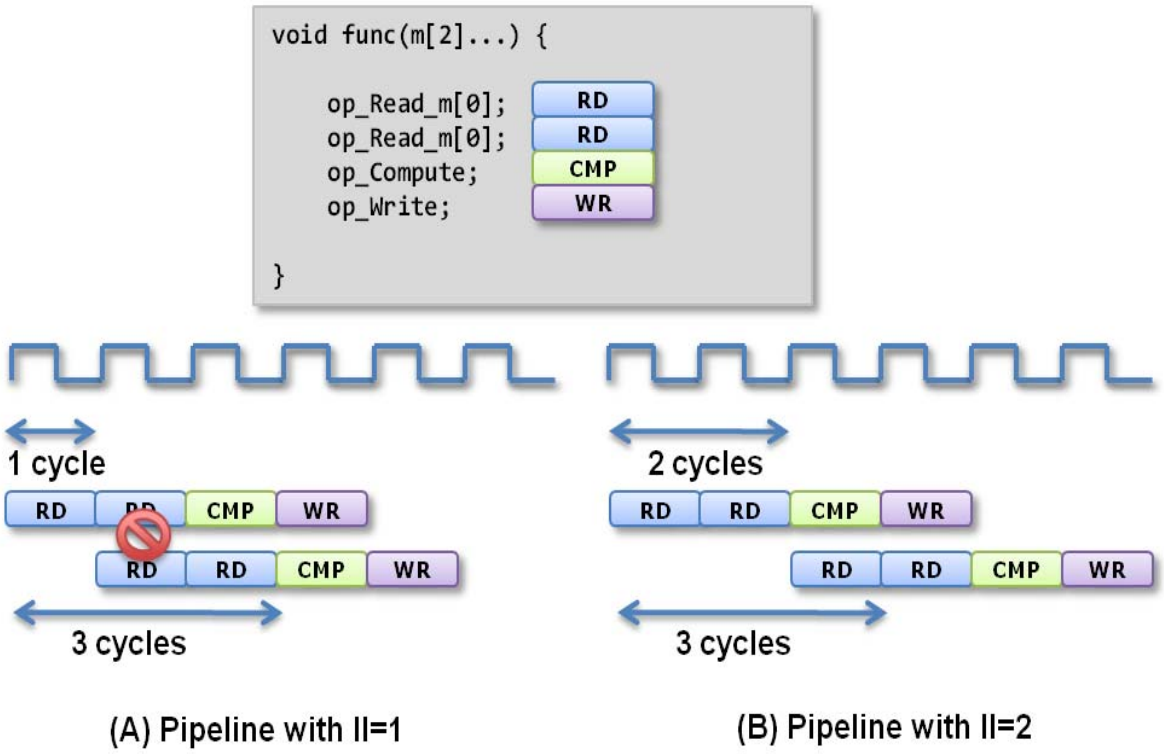

Figure 2-61: Pipeline Resource Contention

The function can be pipelined, as shown in Figure 2-62(B), by increasing the initiation interval of the pipeline. The initiation interval is the number of cycles between new input reads.

In Figure 2-61(A) the initiation interval is 1 because a new operation is performed every clock cycle. In Figure 2-61(B) an initiation interval of 2 is used - there is no longer any resource contention on the input port - and the function can be successfully pipelined.

Note: Resource contentions can occur due to reads and writes on ports, access to limited resources (for example, if only 1 multiplier is available) or reads and writes on arrays mapped to a RAM or FIFO.

Review sections "Array Optimizations" and "Controlling Hardware Resources" for more details on analyzing resource contention if pipelining fails to achieve the desired initiation interval. The resource contention problem in Figure 2-61 could also be solved by using a dual-port RAM for array "m[2]", allowing both reads to be performed in the same clock cycle.

Function pipelining can be applied using the Tcl command

```
set_directive_pipeline function_foo
```

or applied using the directives tab in the GUI. Pipelining is applied hierarchically to the specified function: all sub-functions in the hierarchy below the specified function are automatically (and recursively) inlined and any loops in the hierarchy are automatically unrolled.

The default operation for function pipelining is to create a pipeline which runs forever and never ends. In some cases, it is desirable to have a pipeline which can be "emptied" or "flushed" and the option -flush is provided to perform this. When a pipeline is "flushed" the pipeline stops reading new inputs when none are available (as determined by a data valid signal at the start of the pipeline) but continues processing, shutting down each successive pipeline stage, until the final input has been processed through to the output of the pipeline.

Latency Constraints

High-Level Synthesis supports the use of latency constraints upon a function. When a maximum and/or minimum constraint is placed on the function, High-Level Synthesis will try to ensure all operations in the function complete within the range of clock cycles specified.

Latency constraints can be applied to a function as shown in Figure 2-56 or they can be specified using the set_directive_latency command.

```
set_directive_latency -min 3 -max 5 function_foo
```

If High-Level Synthesis is unable to meet a latency constraint it will allow the timing in one of the clock cycles to be violated, as described in “Clocks, Timing & RTL output”. High-Level Synthesis will produce a design with the minimum timing violation to facilitate a strategy of meeting timing using downstream logic synthesis. If the timing violation is too great to be met using logic synthesis, review the techniques in the “Logic Structure Optimizations” chapter to reduce logic delays.

To confirm that all latency constraints have been satisfied, review the constraint report.

Function Interface Protocol

High-Level Synthesis provides the capability of automatically creating a function interface protocol. When a function is synthesized each of the function parameters, any function return value and any global variables accessed by the function are implemented as input or output ports in the final RTL design. In addition to these ports, High-Level Synthesis can synthesize function control ports which allow the RTL implementation to be more easily integrated into a surrounding system.

The interface protocol provides an input start signal ("ap_start") which must be set to logic 1 before the function will begin execution, an output signal to indicate when the function has completed all operations ("ap_done") and an output idle signal ("ap_idle") to indicate that no operations are currently being performed by the function.

The ability to automatically add a function level interface protocol means the implementation details of the protocol can be omitted from the source code description, allowing it to remain a high-level specification of the algorithm.

An interface protocol can be applied to any function in the hierarchy but it is recommended to only apply an interface protocol to the top-level function and allow High-Level Synthesis to schedule the most optimum communication between sub-functions.

A complete description of the function interface protocol is provided in the "Interface Management" chapter and a detailed waveform diagram of the protocol is shown in Figure 2-33.

Loop Optimizations

Within functions, C language descriptions are typically implemented as a series of loops. Understanding how loops are implemented in HLS, can be optimized and the impact of loop hierarchy is crucial to achieving optimal performance at the RT level.

A complete list of the directives which can be applied to loops can be seen in the GUI (Figure 2-62):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select a loop, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

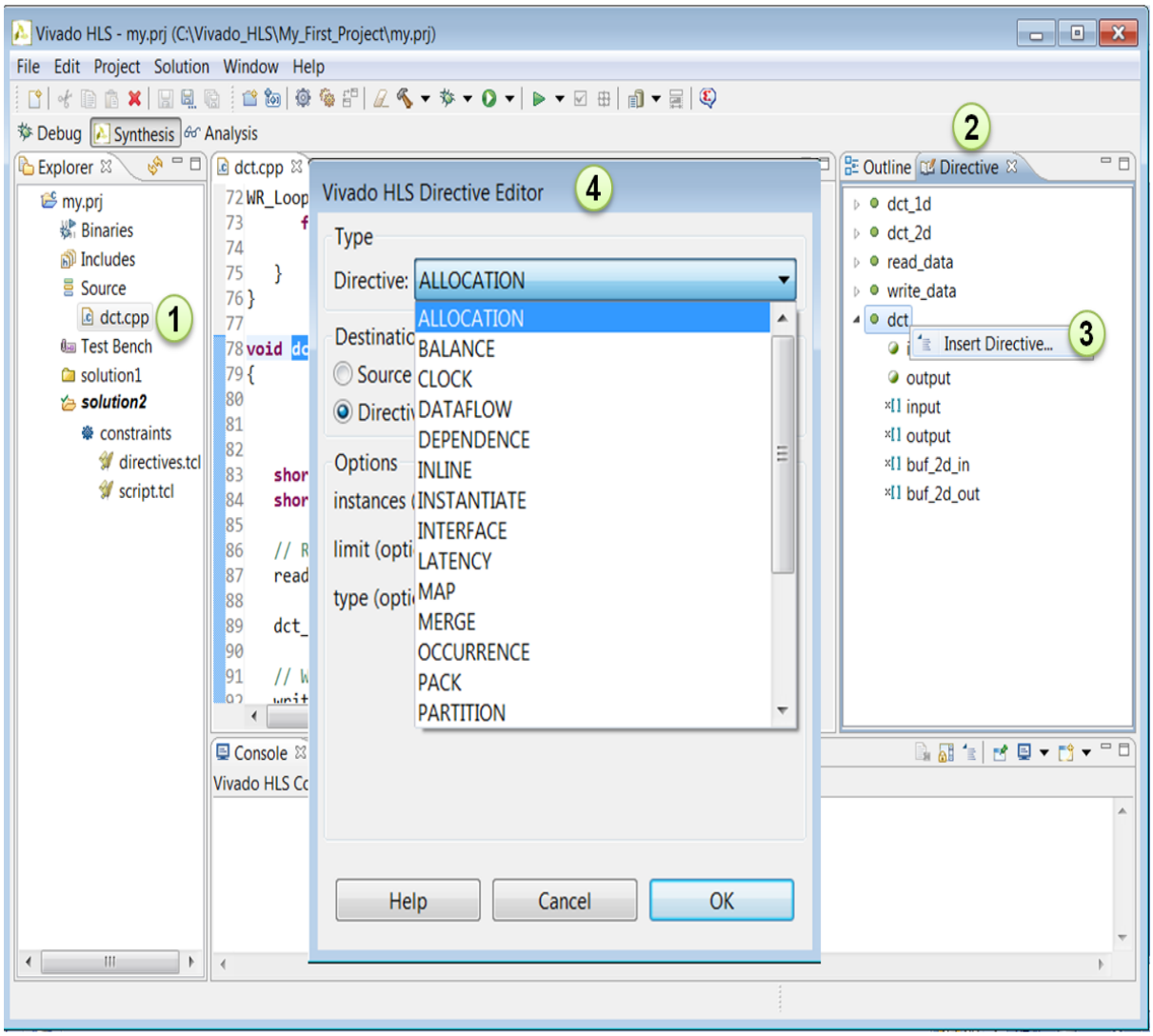

Figure 2-62: GUI Function Directives

IMPORTANT: *Not all of the directives which can be applied to loops are related to the optimization of loops.*

For example, when applied to a loop, the BALANCE directive is applied to the logic structures created with the loop. Logic optimizations are discussed in a subsequent chapter.

Table 2-9 lists the optimizations which can be performed on loops and the order in which the optimizations are discussed in this chapter.

Table 2-9: Loop Level Optimizations

GUI Directive	Description
Unrolling	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
Merging	Merge consecutive loops to reduce overall latency, increase sharing and optimization.
Flattening	Allows nested loops to be collapsed into a single loop with improved latency and logic optimizations
Dataflow	Allows sequential loops to operate concurrently
Pipelining	Used to increase throughput by performing concurrent operations
Dependence	Used to provide additional information which can be used to overcome loop-carry dependencies.
Tripcount	Provides user override of iteration analysis
Latency	Specify a cycle latency for the loop operation

Unrolling Loops

By default loops are kept rolled in High-Level Synthesis. That is to say that the loops are treated as a single entity: all operations in the loop are implemented using the same hardware resources for iteration of the loop.

High-Level Synthesis provides the ability to unroll or partially unroll for-loops.

Figure 2-63 shows both the powerful advantages of loop unrolling and the implications which must be considered when unrolling loops. The example in Figure 2-63 assumes the arrays a[i], b[i] and c[i] are mapped to RAMs. If the arrays were not mapped to sequential elements, the number of cycles in the example would be determined by the combinational delay of the multiplier.

The first conclusion which can be drawn from Figure 2-63 is how easy it is to create many different implementations by the simple application of loop unrolling.

```
void top(…) {

 ...
 for_mult:for (i=3;i>=0;i--) {
      a[i] = b[i] * c[i];
  }
 ...
}
```

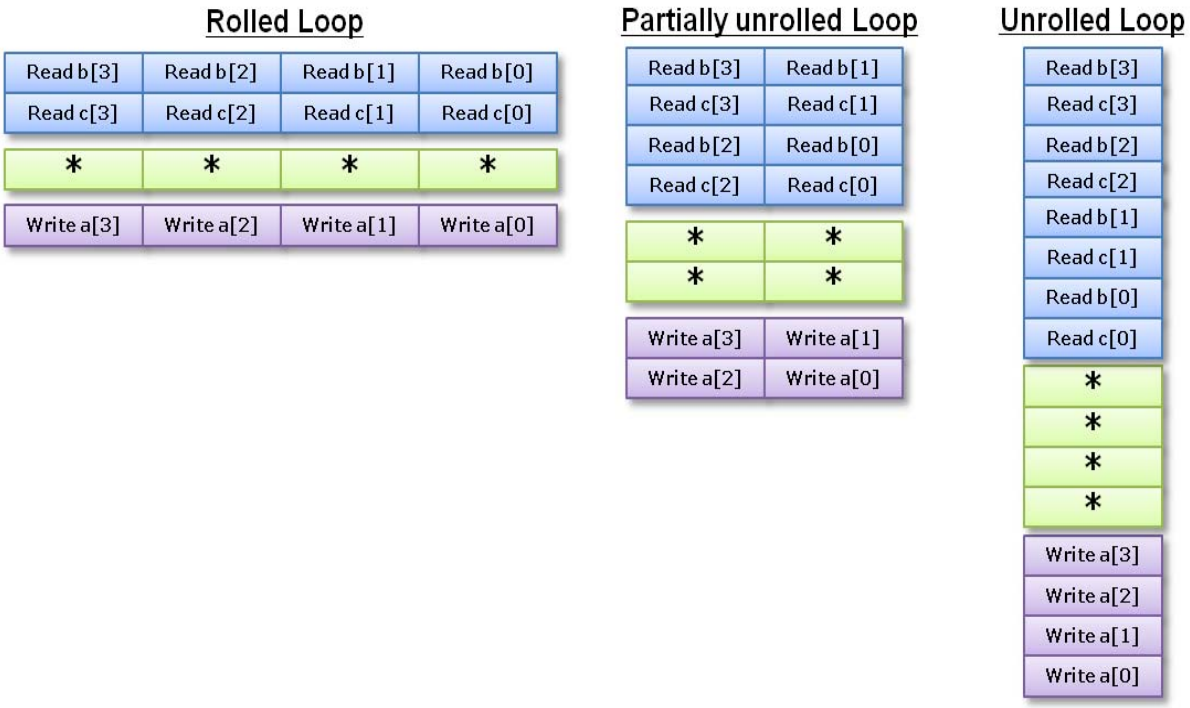

Figure 2-63: Loop Unrolling Details

- **Rolled Loop**: When the loop is rolled, each iteration will be performed in a separate clock cycle. This implementation takes four clock cycles, only requires one multiplier and each RAM can be a single port RAM.
- **Partially Unrolled Loop**: In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: twice the throughput and half the latency of the rolled loop version.
- **Unrolled loop**: In the fully unrolled version the entire loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Since quad-port RAMs are not common, this

implementation may require the arrays be implemented as register arrays rather than RAMs, or that array partitioning and re-shaping be used.

It is safe to say that depending on how the arrays are implemented (some or all mapped to RAMs) and the delay of the multiplier, there could be many more possible implementations of this simple example.

Loop unrolling can be performed using the GUI as shown in Figure 2-56 by applying directives to individual loops in the design. Alternatively, loop directives can be applied to all for-loops in a function by applying the unroll directive to the function itself as shown in Figure 2-56. The Tcl command can also be used to unroll specific loops:

```
set_directive_unroll -skip_exit_check -factor 2 top/for_mult
```

The set_directive_unroll command can only be applied to loops which are labeled, as shown in Figure 2-63, unless the directive is applied as a pragma inserted into the source code (which would apply to all versions of the code).

Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count. High-Level Synthesis will automatically add any exit checks to ensure partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 will effectively transform the code to look like the following example where the "break" construct is used, and implemented in the RTL, to ensure the functionality remains the same:

```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= N) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

Since N is a variable, High-Level Synthesis may not be able to determine its maximum value (it could be driven from an input port). If it is known that the unrolling factor, 2 in this case, is an integer multiple of the maximum iteration count N, the -skip_exit_check option can be used to remove the exit check. The effect of unrolling can now be represented as:

```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

This helps minimize the area and simplify the control logic.

Unrolling Loops in C++ Classes

When loops are used in C++ classes, care should be taken to ensure the loop induction variable is not a data member of the class as this prevents the loop for being unrolled.

In this example, loop induction variable "k" is a member of class "foo_class".

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
  pe_mac<T0, T1, T2> mac;
public:
  T0 areg;
  T0 breg;
  T2 mreg;
  T1 preg;
   T0 shift[N];
  int k;              // Class Member
   T0 shift_output;
  void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
  {
Function_label0:;
#pragma AP inline off
  SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll// Loop will fail UNROLL
    if (k > 0)
      shift[k] = shift[k-1];
    else
      shift[k] = data;
  }

  *dataOut = shift_output;
  shift_output = shift[N-1];
  }

  *pcout = mac.exec1(shift[4*col], coeff, pcin);
};
```

For High-Level Synthesis to be able to unroll the loop as specified by the UNROLL pragma directive, the code should be re-written to remove "k" as a class member.

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
  pe_mac<T0, T1, T2> mac;
public:
  T0 areg;
  T0 breg;
  T2 mreg;
  T1 preg;
   T0 shift[N];
   T0 shift_output;
  void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
  {
Function_label0:;
  int k;              // Local variable
#pragma AP inline off
```

```
  SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll// Loop will unroll
    if (k > 0)
      shift[k] = shift[k-1];
    else
      shift[k] = data;
  }

  *dataOut = shift_output;
  shift_output = shift[N-1];
  }

  *pcout = mac.exec1(shift[4*col], coeff, pcin);
};
```

Merging Loops

All rolled loops imply and create at least one state in the design Finite-State-Machine (FSM). When there are multiple sequential loops this can sometimes create additional unnecessary clock cycles and prevent further optimizations.

Figure 2-64 shows a simple example where a seemingly intuitive coding style has a negative impact on the performance of the RTL design.

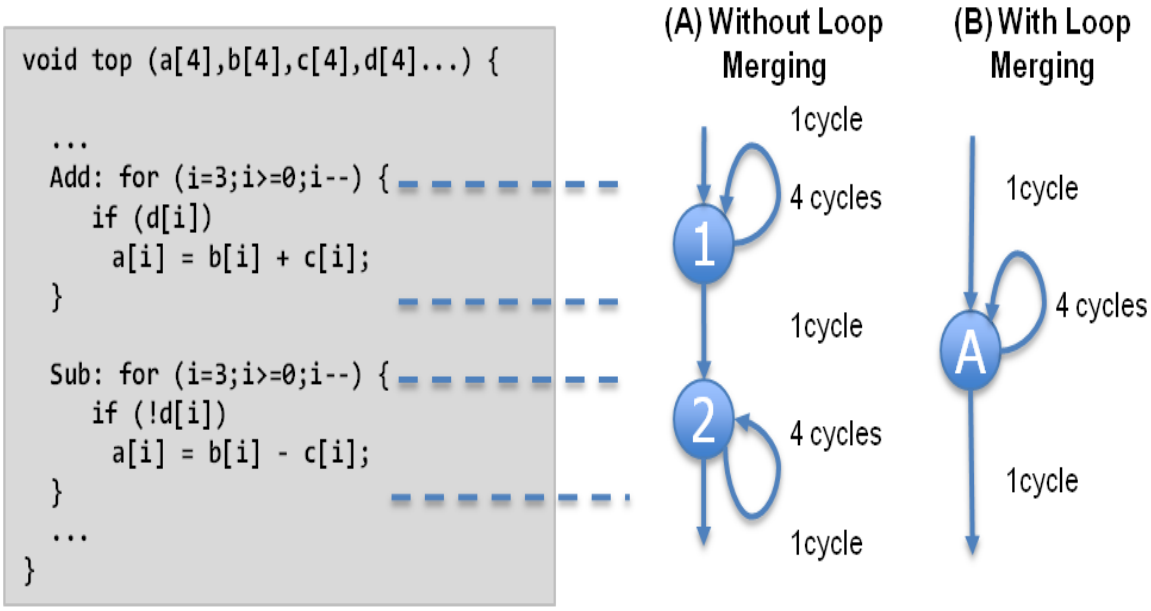

Figure 2-64: Loop Merging

Figure 2-64(A) shows how by default, each rolled loop in the design creates at least one (but perhaps more, depending on the number of operations to be performed) state in the

FSM3. Moving between those states costs clock cycles: assuming each loop iteration requires one clock cycle, it take a total of 11 cycles to execute both loops:

- 1 clock cycle to enter the ADD loop.
- 4 clock cycles to execute the add loop.
- 1 clock cycle to exit ADD and enter SUB.
- 4 clock cycles to execute the SUB loop.
- 1 clock cycle to exit the SUB loop.
- For a total of 11 clock cycles.

In this simple example it should become obvious that an else branch in the ADD loop would also solve the problem but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages, such as allowing colleagues in the same design team to more easily understand a very complex algorithm. (Adding statements to perform the same merge operation, such as multiple if-else statements, may make the code very unreadable).

High-Level Synthesis provides a feature to automatically merge loops. Using the directives tab in the GUI to add a MERGE directive to the function (or a region which contains both loops) would instruct High-Level Synthesis to merge the loops and create a control structure similar to that shown in Figure 2-64(B) which requires only 6 clocks to complete.

Currently, loop merging in High-Level Synthesis has the following restrictions:

- If loop bounds are all variables, they must have the same value.
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO reads: merging would change the order of the reads and reads from a FIFO or FIFO interface must always be in sequence.

Loop merging can be performed at the command line using the set_directive_loop_merge command (on a function as shown below or on a labeled loop or region) or it can be performed using the GUI to apply the MERGE directive.

```
set_directive_loop_merge top
```

Flattening Nested Loops

In a similar manner to the consecutive loops discussed in the previous section, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop.

In the small example shown here, this implies 200 extra clock cycles to execute loop "Outer".

```
void foo_top { a, b, c, d} {
  ...
  Outer: while(j<100)
    Inner: while(i<6)// 1 cycle to enter inner
      ...
      LOOP_BODY
      ...
    }    // 1 cycle to exit inner
  }
  ...
}
```

In addition, nested loops prevent the outer loop from being pipelined, as discussed in the next section on "Loop Dataflow Pipelining".

High-Level Synthesis provides the set_directive_loop_flatten command to allow labeled perfect and semi-perfect nested loops to be automatically flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop.

- **Perfect loop nest**: only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.
- **Semi-perfect loop nest**: only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

When the directive is applied to a set of nested loops it should be applied to the inner most loop which contains the loop body.

```
set_directive_loop_flatten top/Inner
```

Loop flattening can also be performed using the directive tab in the GUI, either by applying it to individual loops or applying it to all loops in a function by applying the directive at the function level.

Loop Dataflow Pipelining

Dataflow pipelining can be applied to loops in similar manner as it can be applied to functions. It allows loops which are sequential in nature to operate concurrently at the RTL. Dataflow pipelining should be applied to a function, loop or region which contains all function or all loops: do not apply on a scope which contains a mixture of loops and functions.

Figure 2-65 shows the advantages dataflow pipelining can produce when applied to loops.

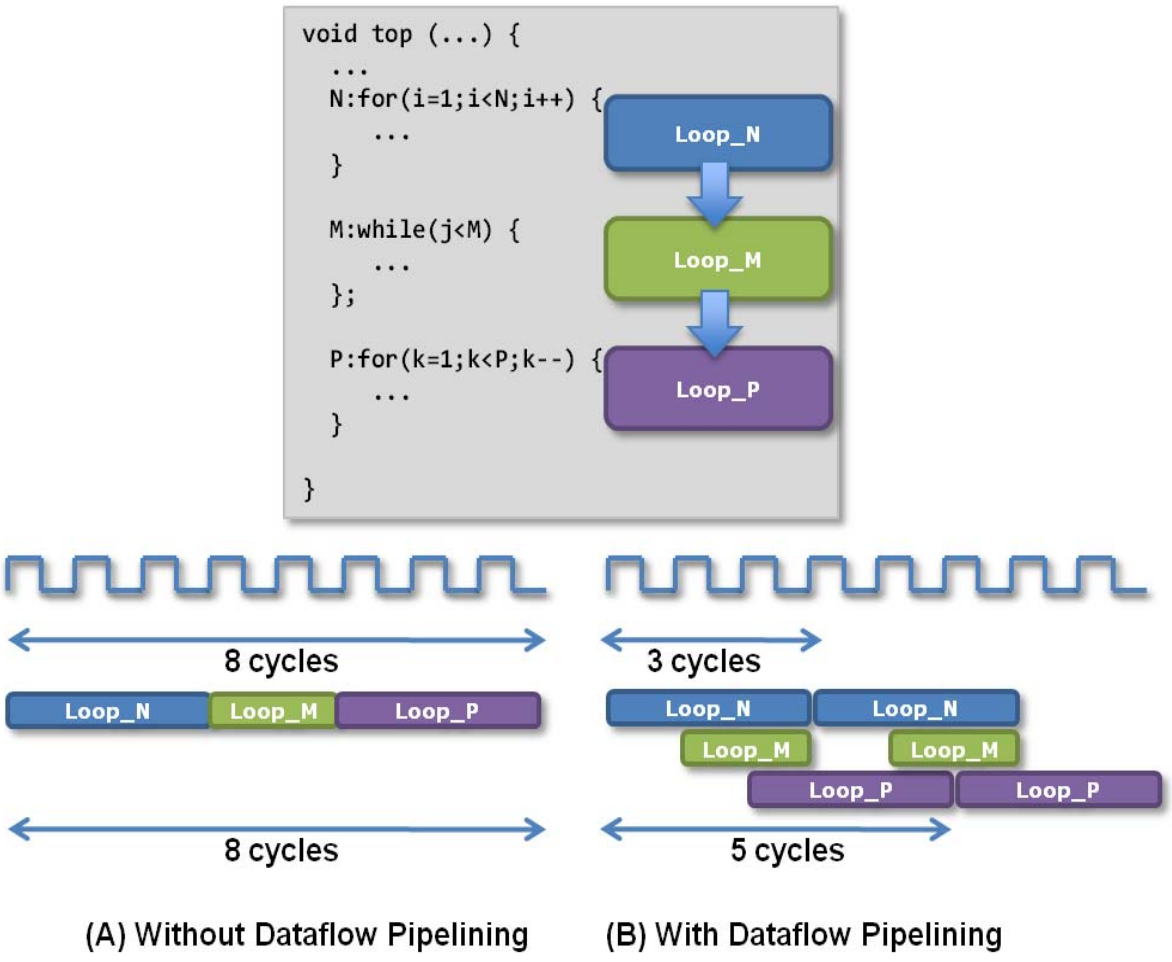

Figure 2-65: Loop Dataflow Pipelining

Without dataflow pipelining, loop N must execute and complete all iterations before loop M can begin. The same applies to the relationship between loops M and P. In this example, it is 8 cycles before loop N can start processing the next value and 8 cycles before an output is written (assuming the output is written when loop P finishes).

With dataflow pipelining, these loops can be allowed to operate in parallel, accepting new inputs every 3 cycles and outputting a value every 5 cycles: using the same hardware resources. High-Level Synthesis automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next.

The channels between the loops are implemented as either ping-pong buffers or FIFOs.

- If the variable is an array the channel is implemented as a ping-pong buffer using standard memory accesses (with associated address and control signals).
- For all other variable types, and streaming arrays, the channel is implemented as a FIFO, which uses less hardware resources (no address generation), but requires the data is accessed sequentially.

IMPORTANT: *To use dataflow pipelining the variables must be produced by one loop and consumed by only one other loop.*

In addition to using the GUI directive tab, dataflow pipelining can be specified using the set_directive_dataflow command. When the directive is specified in a region High-Level Synthesis will seek to improve the concurrency of the loops within it. For the example shown in Figure 2-65 the following command would perform loop dataflow pipelining on loops "Loop_N", "Loop_M" and "Loop_P".

```
set_directive_dataflow -interval 2 top
```

The –interval option can be used to specify exactly how many cycles there will be between the start of one loop implementation and the next. By default High-Level Synthesis will try to minimize this time. Ideally High-Level Synthesis will try to have them all start of the same clock edge and execute in parallel, but data dependencies will typically prevent this. For example, if an interval of 3 is specified, there would be 3 cycles between the start of loop implementation in Figure 2-65(B).

The number of elements in the channel (memory) is defined by the maximum size of the consumer or producer array size but High-Level Synthesis provides a means to specifying a default channel depth (refer to "Configuring " below).

Configuring Default Channels

The default channel used between loops can be specified using the config_dataflow command. Configuration commands allow a default operation to be set for a solution. This command allows the default channel size and implementation to be set for all channels in a design.

```
config_dataflow -default_channel (fifo | *pingpong*) -size <FIFO size>
```

The size of a channel is defined by the maximum size of the consumer or producer array. In some cases this may be overly conservative. The -size option provides a means for the user to override the default behavior.

If an array parameter is specified to use a FIFO channel, the array is automatically converted to a streaming type (refer to "Array Streaming" for an explanation of streaming). If the default channel type is FIFO but a specific array has been specified as non-streaming using set_directive_array_stream command, the channel implementation for that array will default to a ping-pong channel. (An explicit directive overrides a configuration).

Loop Pipelining

In a C language description the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the loop is complete. An RTL design can execute multiple operations concurrently and it is often desirable that the RTL be implemented to perform in this manner.

Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in Figure 2-66. The default sequential operation is shown in Figure 2-66(A) where there are 3 clock cycles between each input read and it requires 8 clock cycles before the last output write is performed.

In the pipelined version of the loop, shown in Figure 2-66(B), a new input sample is read every cycle and the final output is written after only 4 clock cycles: substantially improving both the throughput and latency while using the same hardware resources, since the only changes to the design are in the control logic.

The number of cycles between new input reads is called the pipeline initiation interval and is specified by the user but defaults to 1 if not specified.

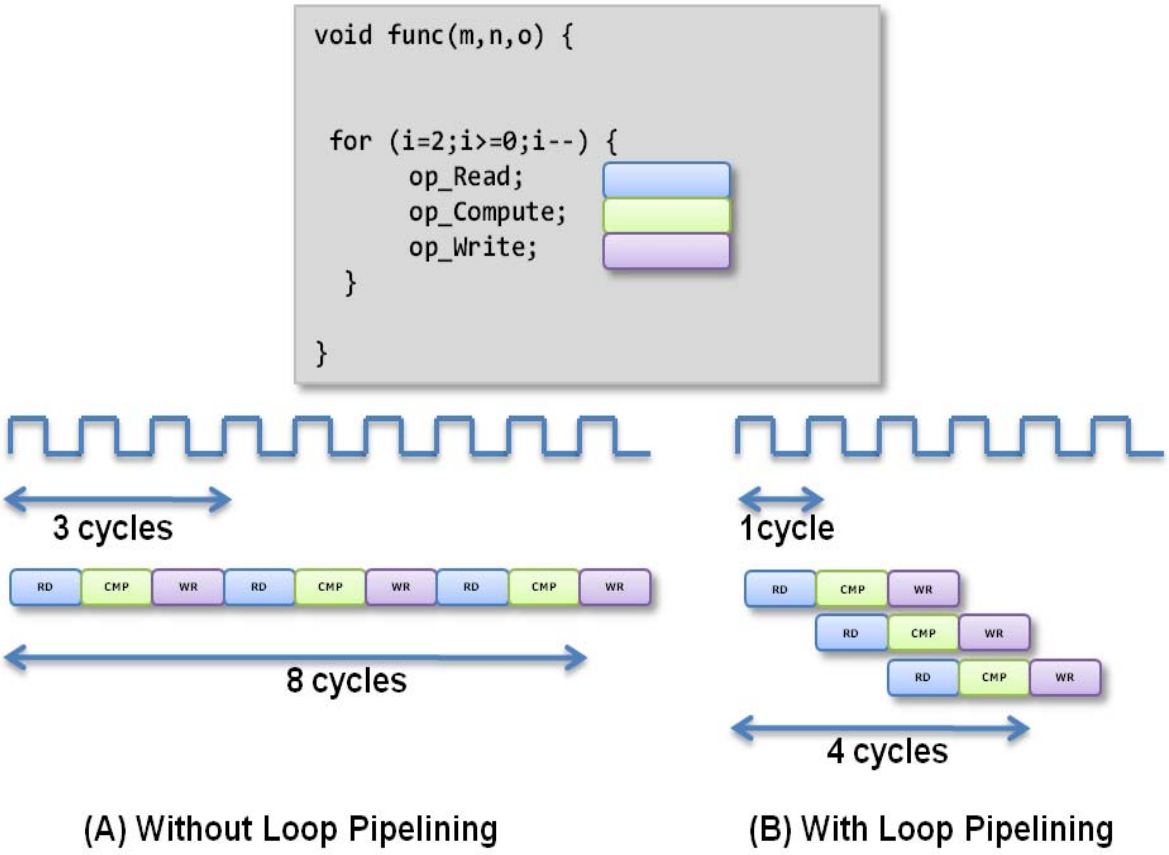

Figure 2-66: Loop Pipelining

Loop pipelining can be prevented due to resource contention, as shown for pipelined functions back in Figure 2-61, and data dependencies.

Only the inner-most loop in a series of nested loops can be pipelined, however pipelining will automatically flatten any loops in the hierarchy below the pipelined region.

Data dependencies are a much harder issue to resolve and often require changes to the source code. A scalar data dependency could look like:

```
while (a != b) {
   if (a > b) a -= b;
   else b -= a;
 }
```

Obviously, the next iteration of this loop can not start until the current iteration has calculated the updated the values of a and b (Figure 2-67).

Figure 2-67: Scalar Dependency

Dependencies are common with memory accesses:

```
for (i = 1; i < N; i++)
  mem[i] = mem[i-1] + i;
```

In this case, the next iteration of the loop must wait until the current iteration updates the content of the array (Figure 2-68).

Figure 2-68: Memory Dependency

If the result of the previous loop iteration must be available before the current iteration can begin, loop pipelining is not possible. If High-Level Synthesis cannot pipeline with the specified initiation interval it will automatically increase the initiation internal (in effect, pulling the next iteration into the current iteration to remove the dependency). If it cannot pipeline at all, as shown by the above examples in Figure 2-67 and Figure 2-68, it halts pipelining and proceeds to output a non-pipelined design with reduced throughput.

Loop pipelining can be specified on loops using the directives tab in the GUI and labeled loops can be pipelined using the Tcl command as shown in this example:

```
set_directive_pipeline -II 5 -enable_flush foo/sum_loop
```

The initiation interval of the pipeline is specified as 5 in this example, meaning a new input will be read every 5 clock cycles and flushing is enabled. Flushing means the pipeline is constructed such that it can be shutdown and cleanly emptied. The default, with no flushing, creates a pipeline which will run continuously until the design is reset.

Loop Carry Dependencies

Loop pipelining can be prevented by loop carry dependencies as explained in the previous section ("Loop Pipelining"). However, under certain complex scenarios automatic dependence analysis can be too conservative and fail to filter out false dependencies.

For instance:

```
void foo(int A[3*N], int x)
{
   LF: for (i = 0; i < N; i++)
           A[i+x] = A[i] + i; // User knows that 2*N > x >= N
}
```

In the above example, the High-Level Synthesis does not have any knowledge about the range of input parameter "x" and will conservatively assume that there is always a dependence between the write to "A[i+x]" and the read from "A[i]" and schedule the loop iterations sequentially.

To overcome this deficiency, the user can specify the dependence directive to provide High-Level Synthesis with additional information about the loop-carried dependencies on one or multiple variables. Here we can inform the tool that no loop-carried dependencies would exist if we know in advance that x is no less than N (and no greater than 2*N).

```
set_directive_dependence -variable x -type inter -dependent false foo/LF
```

When specifying dependencies there are two main types:

- **Inter**: specifies the dependency is between different iterations of the same loop. If this is specified as false it will allow High-Level Synthesis to perform operations in parallel if the loop is unrolled or partially unrolled and prevents such concurrent operation when specified as false.
- **Intra**: specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration. When intra dependencies are specified as false High-Level Synthesis may move operations freely within the loop, increasing their mobility and potentially improving performance or area. Obviously when the dependency is specified as true, the operations must be performed in the order specified.

Loop Iteration Control

High-Level Synthesis performs analysis to automatically determine the maximum possible iteration of a loop, however it is not possible for High-Level Synthesis to determine the actual maximum iteration of a loop.

In the following example, the maximum iteration of the for-loop is determined by the value of input "num_samples". High-Level Synthesis can determine that the maximum possible

value of this variable is 15 (it is of type uint4), but it cannot know that the actual value of "num_samples" is, for example, never above 9 due to an external constraint.

```
void foo (uint4 num_samples, ...);

void foo (num_samples, ...) {
   int i;
   ...
   loop_1: for(i=0;i< num_samples;i++) {
  ...
       result = a + b;
   }
}
```

If the latency or throughput of the design is dependent on a loop with a variable index, High-Level Synthesis will not be able to report the correct values for throughput or latency. High-Level Synthesis will report the latency of the loop as being unknown (represented in the reports by a question mark "?").

Specifying the loop iterations, or tripcount, ensures the report contains valid numbers. This can be specified using the TRIPCOUNT directive which can be applied via the GUI using the directives tab or specified on labeled loops using the Tcl command line:

```
set_directive_loop_tripcount -min 3 -max 8 -avg 5 foo/loop_1
```

The -max option tells High-Level Synthesis the maximum number of iterations the loop will ever iterate. The -min option specifies the minimum number of iterations which will be performed and the -avg option specifies an average tripcount. This ensures the High-Level Synthesis reports accurately reflect the maximum and minimum design throughput.

The tripcount directive does not impact the results of synthesis. The tripcount values are only used for reporting, where they help to ensure the reports generated by High-Level Synthesis show meaningful ranges for latency and throughput.

Loop Latency

The maximum and minimum latency for a loop can be specified as a constraint. This is a means of ensuring the performance targets are met and a powerful way to control how the resources in a loop are used.

By default High-Level Synthesis will seek to meet timing and then minimize latency. If timing cannot be met, latency will be extended until timing can be met. If there is a latency constraint which prevents latency being extended, High-Level Synthesis will allow local timing violations as detailed in the "Clocks, Timing & RTL output" section. Given the latency achieved with the default settings:

- Setting a maximum latency which is less than this will cause High-Level Synthesis to work harder to meet the lower latency value by trying to improve operator sharing and chaining.
- Setting a minimum latency which is higher than this will allow High-Level Synthesis to take more clock cycles to complete the operations, allowing increased sharing and potentially meeting timing on any paths which are currently failing.

Latency constraints are applied on each loop separately via the GUI or using the set_directive_latency command using the loop label.

Array Optimizations

The memory configurations in a design have a great impact on the performance and area of the overall design. Arrays in a C language description are typically mapped to memories and so the optimizations performed on arrays have a great impact on both area and performance.

A complete list of the directives which can be applied to arrays can be seen in the GUI (Figure 2-69):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select an array, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

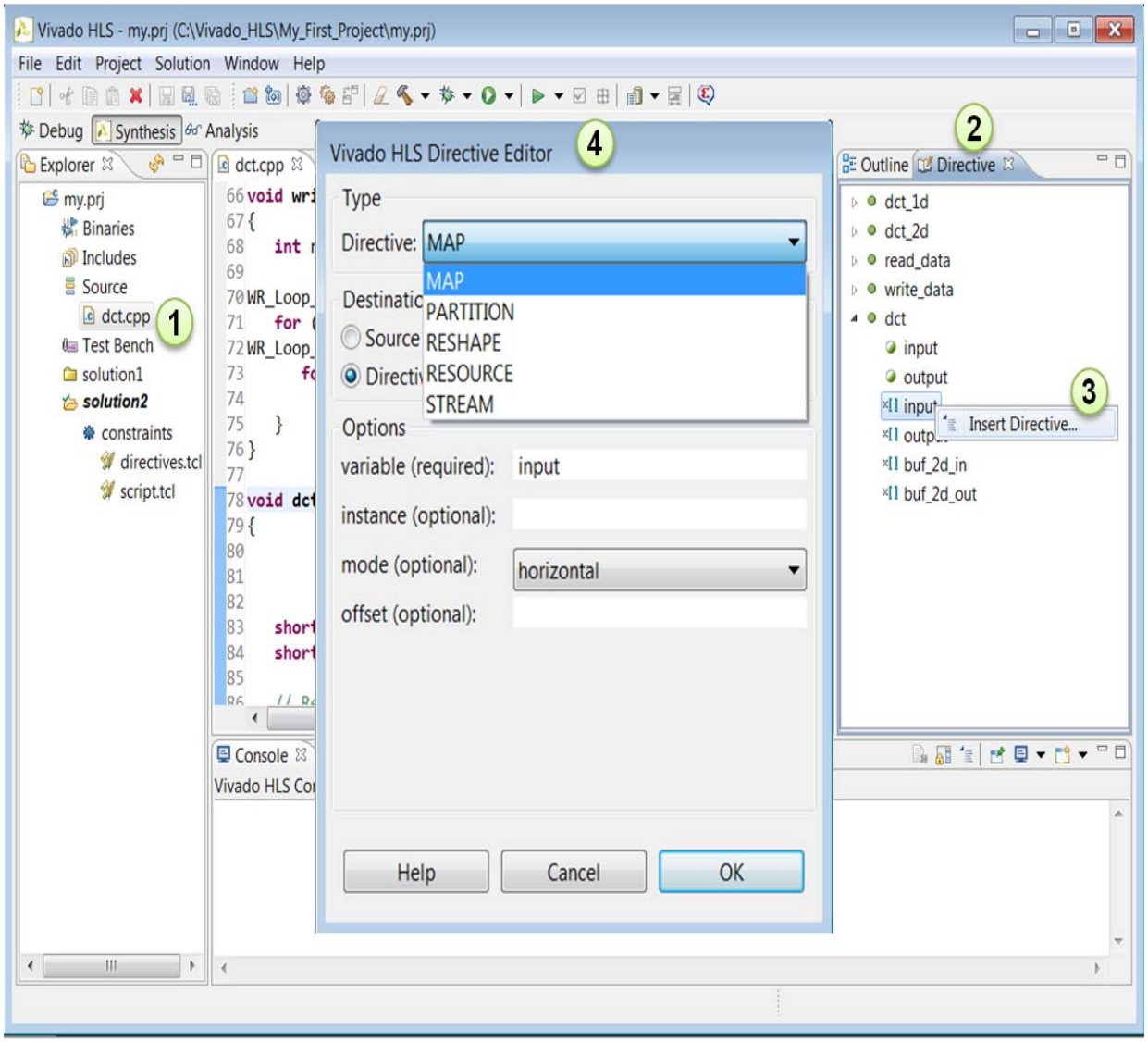

Figure 2-69: GUI Array Directives

Table 2-10 lists the optimizations which can be performed on arrays and the order in which they are discussed in this chapter.

Table 2-10: Array Optimizations

GUI Directive	Description
Resource	Specify which hardware resource (RAM component) a array maps to.
Map	Reconfigures array dimensions by combining multiple smaller arrays into a single large array to help reduce RAM resources and area.
Partition	Control how large arrays are partitioned into multiple smaller arrays to reduce RAM access bottleneck. Also used to ensure arrays are implemented as registers and not RAMs.
Reshape	Can reshape an array from one with many elements to one with greater word-width. Useful for improving RAM accesses without using many RAMs.
Stream	Specifies that an array should be implemented as a FIFO rather than RAM.

Arrays in a C language description are generally implemented using memory resources at the RTL. This chapter explains how arrays can be implemented using specific RAMs or ROMs, how they can be transformed (split horizontally, split vertically, aggregated or combinations of these operations) to ensure they map efficiently into the available memory resource and how they can be decomposed into individual registers.

Arrays which are specified as arguments to the top-level function are synthesized in a slightly different manner. In this case it is assumed the memory resource is outside the design and the array is synthesized into ports which can access this resource. The same transformations can be performed upon such arrays: the result of synthesis will simply be ports which can access the transformed array.

Array Initialization & Reset

High-Level Synthesis supports the initialization of arrays in the source code. The initialization value of arrays is replicated in the RTL and in the FPGA, where the bitstream is used to ensure the RAM (or registers if the array is partitioned) is initialized to the same values when the device is powered-up.

The following code example shows:

- An example where the array is initialized from file fir_coef.h.

- An example where the const qualifier is used: High-Level Synthesis will know the array should be implemented as a ROM.

If the const qualifier is not used High-Level Synthesis can generally detect when arrays are only read from and thus should be implemented as a ROM, however it is always best to specify explicitly which memory resource should be used to implement the array. Resource selection is discussed in the next section.

```
typedef intcoef_t;
typedef intdata_t;
typedef intacc_t;

     data_t fir (
  data_t x
      ) {
      static data_t shift_reg[N];
      acc_t acc;
      int i;

     const coef_t c[N+1]={
#include "fir_coef.h"
      };

      acc=0;
      mac_loop: for (i=N-1;i>=0;i--) {
         if (i==0) {
           acc+=x*c[0];
           shift_reg[0]=x;
         } else {
           shift_reg[i]=shift_reg[i-1];
           acc+=shift_reg[i]*c[i];
         }
      }
      return=acc;
     }
```

Arrays will be initialized to the values specified in the source code at power-up, however subsequent applications of the reset port to the device will not return the array to this initial power-on state: the reset of arrays is not supported through any reset added by High-Level Synthesis.

If arrays must be returned to some initial (reset) state when an external signal is applied, this must be explicitly coded into the design. An example of this is:

```
...
     const coef_t c_temp[N+1]={
#include "fir_coef.h"
      };
     coef_t c[N+1];

     if (rst_array==1) {
  for (i=0;i<N;i++) {
    c[i] = c_temp[i];
  }
     }
```

```
...
```

This however may result in two undesirable attributes in the RTL design:

- Unlike a power-up initialization, this type of explicit reset requires that the RTL design iterate through each address in the array/RAM to set the value: this can take many clock cycles if N is large.
- This requires that an additional variable, rst_array in this case, is added to the top-level function interface. This port would be in addition to any reset signal which is added during synthesis.

Memory Resource Selection

If no memory resource is specified for an array, High-Level Synthesis will automatically determine which memory resource (single-port, dual-port, etc.) will be used. The same applies to arrays specified as function arguments on the top-level function: High-Level Synthesis may create an interface to a dual-port memory since it allows higher throughput. This is not guaranteed to be the best choice and so users are encouraged to specify exactly which memory resource each array should map to.

Arrays are mapped to a specific RAM resource using the set_directive_resource command as shown below or this can be specified in the GUI as shown in Figure 2-69 (or inserted as a pragma in the code).

Given the following example with three arrays, one specified as a function parameter and two defined within the function,

```
void foo (in[16], ...) {
   int8  array1[16];
   int12 array2[48];
   ...
  loop_1: for(i=0;i<8;i++) {
    array1[i] = in[i];
    ...
  }
}
```

Interface Resources

The following will specify the type of RAM which supplies the data to port in[16]:

```
set_directive_resource -core RAM_1P foo in
```

The -core option specifies the RAM core: a complete list of RAM cores is available in the "High-Level Synthesis Library Guide" and can be selected from the RESOURCE directive window in the GUI.

If parameter "in[16]" is specified as having an ap_memory interface, the ports created will match the ports on RAM_1P. If RAM_1P has a chip-enable (CE) port, High-Level Synthesis

will create an interface with a CE port. If the RAM_1P has separate address ports for reading and writing, High-Level Synthesis will create an address port to read the RAM and an address port to write to the RAM. If it has a single address port for both, High-Level Synthesis will create a design with a single address port for both read and write operations.

If port in[16] is specified as an ap_fifo the type of memory resource specified is of less importance since an ap_fifo interface is always the same: data ports with read, write, empty and full ports.

Refer to chapter "Interface Management" for more details on selecting an interface.

Design Resources

Using the same code example above, the following commands specify the type of RAM used to implement arrays "array1" and "array2".

```
set_directive_resource -core RAM_1P foo array1
set_directive_resource -core RAM_2P foo array2
```

In this case, "array1" is mapped to core RAM_1P and "array2" is mapped to core RAM_2P. At this stage, the following must be satisfied:

- RAM_1P must have more than 16 elements (addresses) and each element must be greater than 8-bits, since int8 is an 8-bit datatype.
- RAM_2P must have more than 28 elements and each element must be more than 12-bits (int12 is a 12-bit datatype).

Even if port "in1" has already been specified as communicating with a RAM_1P component outside the function, there is no requirement that "array1" be targeted to the same type of RAM component. High-Level Synthesis will perform any transformation necessary to read from one type of RAM or RAM port and write to another.

Array Mapping

In most technology libraries the RAMs provided have pre-defined sizes (e.g. power-of-2 depth, with 1,8,16-bit words). When there are many small arrays in the original specification, mapping them into a single large array before specifying a target resource may reduce the storage overhead. If each small array gets a separate memory, a lot of memory space is potentially wasted and the design will be unnecessarily large.

The High-Level Synthesis set_directive_array_map command supports two ways of mapping small arrays into a larger one:

- **Horizontal mapping**: this corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.

- **Vertical mapping**: this corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single array with a larger bit-width.

Horizontal Mapping

The following code example has two arrays which would result in two RAM components.

```
void foo (...) {
   int8  array1[M];
   int12 array2[N];
   ...
  loop_1: for(i=0;i<M;i++) {
    array1[i] = ...;
    array2[i] = ...;
    ...
  }
   ...
}
```

Arrays "array1" and "array2" can be combined into a common array, specified as "array3" in the following example:

```
set_directive_array_map -instance array3 -mode horizontal foo array1
set_directive_array_map -instance array3 -mode horizontal foo array2
```

The MAP directive, which can also be specified in the GUI by selecting the individual arrays, transforms the arrays as shown in Figure 2-70.

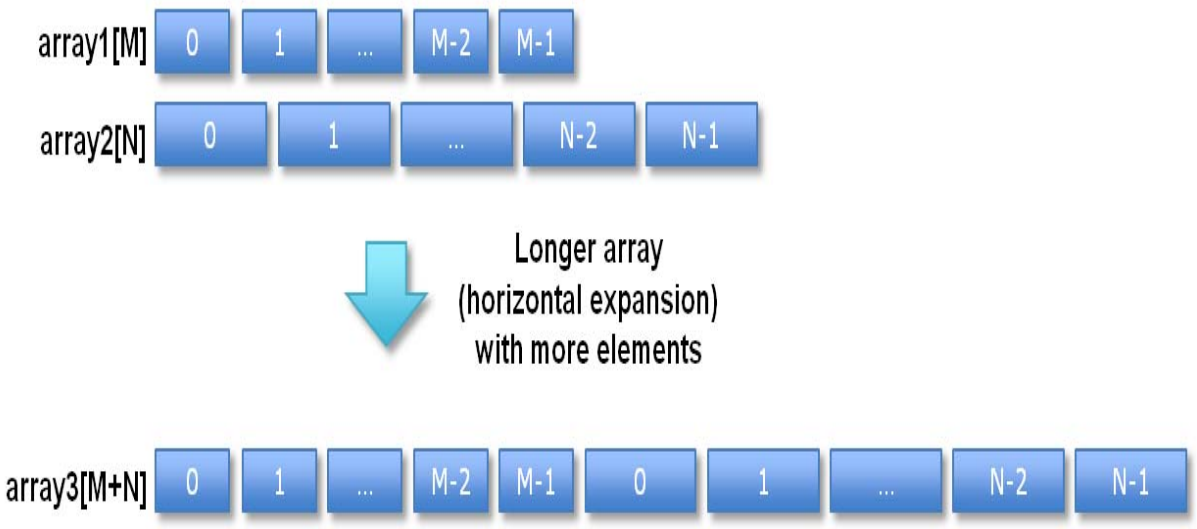

Figure 2-70: Horizontal Mapping

When using horizontal mapping the smaller arrays are mapped into a larger array, starting at location 0 in the larger array, and in the order the commands are issued (in the

High-Level Synthesis GUI it is the order the arrays are specified using the menu). The -offset option can be used to add additional elements between the original arrays.

To repeat the previous example, but reversing the order of the commands (adding "array2" then "array1") and adding an offset, as shown below,

```
set_directive_array_map -instance array3 -mode horizontal foo array2
set_directive_array_map -instance array3 -mode horizontal -offset 2 foo array1
```

would result in the transformation shown in Figure 2-71.

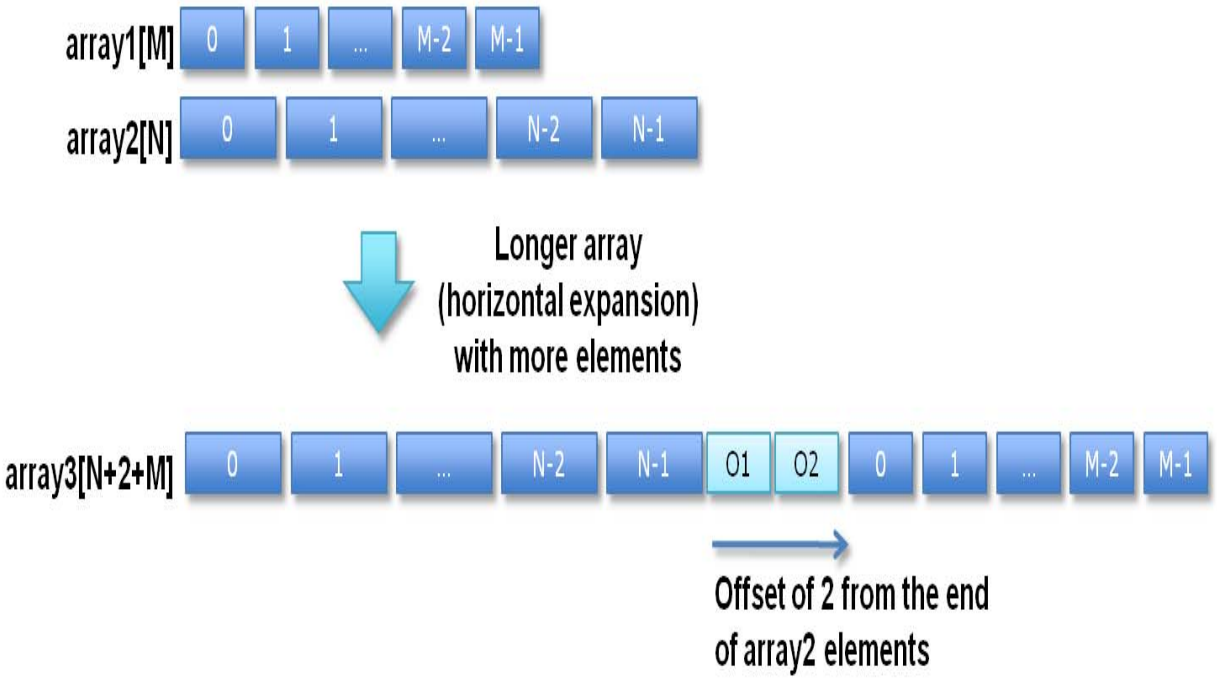

Figure 2-71: Horizontal Mapping with Offset

After mapping the newly formed array, "array3" in the above examples, can be mapped into a single RAM component.

```
set_directive_resource -core RAM_1P foo array3
```

The RAM implementation shown in Figure 2-72 corresponds to the mapping in Figure 2-70 (no offset is used).

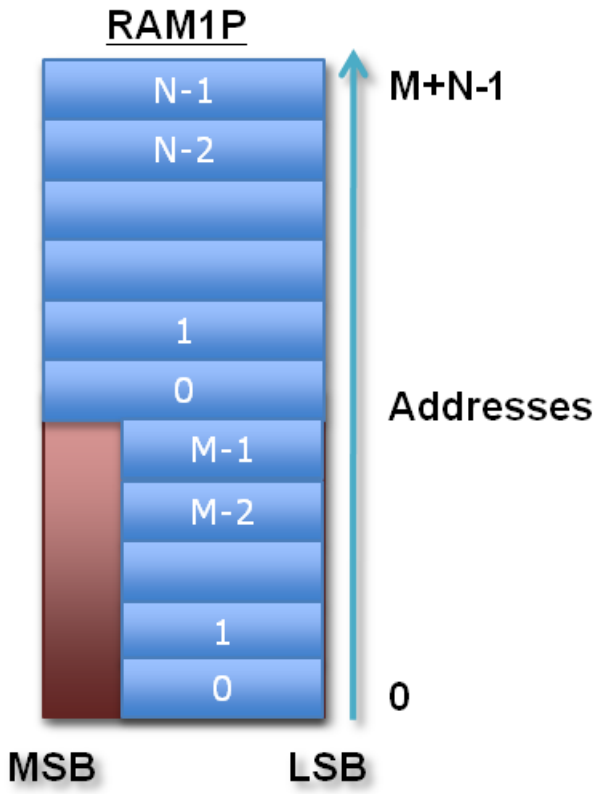

Figure 2-72: Memory for Horizontal Mapping

Although horizontal mapping can result in using less RAM components and hence improve area, it can have an impact on throughput and performance. In the above example both the accesses to "array1" and "array2" in "loop_1" can be performed in the same clock cycle. If both arrays are mapped to the same RAM this will now require a separate access, and clock cycle, for each read operation.

To overcome this limitation, High-Level Synthesis provides vertical mapping.

Vertical Mapping

In vertical mapping, arrays are concatenated by to produce an array with higher bit-widths. Figure 2-73 shows how the same example used in horizontal mapping discussion is transformed when vertical mapping mode is applied.

```
set_directive_array_map -instance array3 -mode vertical foo array2
set_directive_array_map -instance array3 -mode vertical foo array1
```

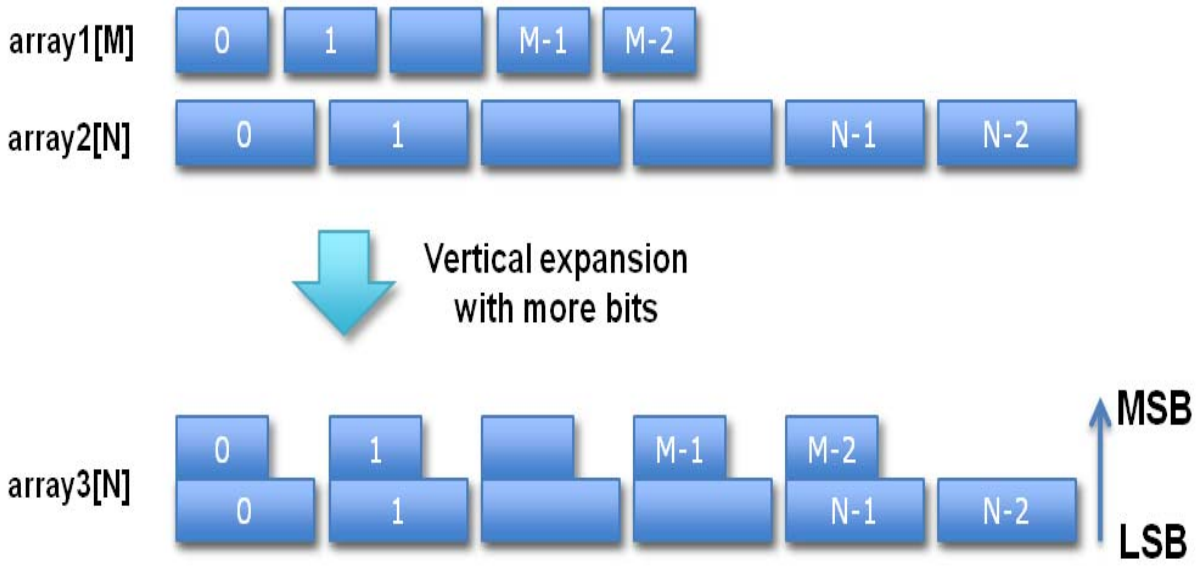

Figure 2-73: Vertical Mapping

In vertical mapping the arrays are concatenated in the order specified by the command, with the first arrays starting at the LSB and the last array specified ending at the MSB. (Note, "array2" was specified first by the set_directive_array_map command. In the High-Level Synthesis GUI it is the order the arrays are specified using the menus).

After mapping the newly formed array, "array3" in the above examples, can be mapped into a single RAM component (Figure 2-74).

```
set_directive_resource -core RAM_1P foo array3
```

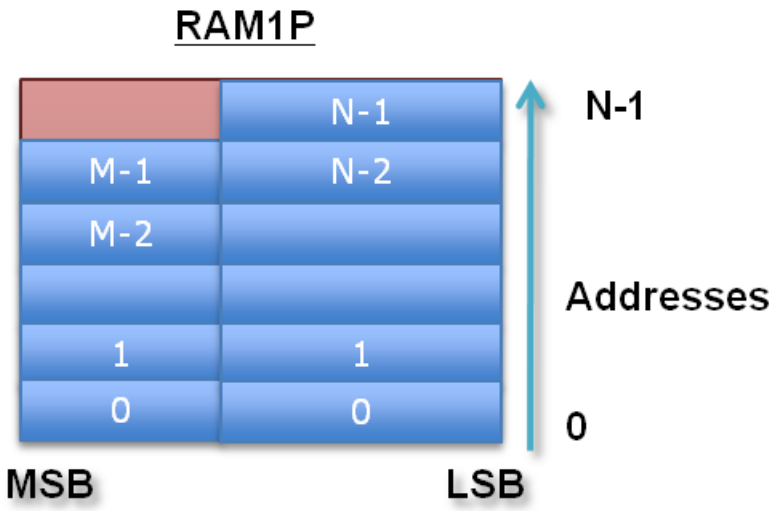

Figure 2-74: Memory for Vertical Mapping

Mapping & Global Arrays

It is possible to map a global array. However the resulting array instance will be global and any local arrays mapped onto this same array instance will become global.

When local arrays of different functions get mapped onto the same target array, then the target array instance becomes global.

When array function parameters are mapped, they need to be parameters of the same function.

Array Partitioning

Arrays can also be partitioned into smaller arrays. Memories only have a limited amount of read ports and write ports which can limit the throughput of a load/store intensive algorithm. The bandwidth can sometimes be improved by splitting up the original array (a single memory resource) into multiple smaller arrays (multiple memories), effectively increasing the number of ports.

Partitioning a larger array into smaller arrays with the set_directive_array_partition command can hence improve the throughput.

High-Level Synthesis provides three types of array partitioning, as shown (Figure 2-75). The various types are specified with the -type option:

- **block**: the original array is split into equally sized blocks of consecutive elements of the original array.

- **cyclic**: the original array is split into equally sized blocks interleaving the elements of the original array.
- **complete**: the default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

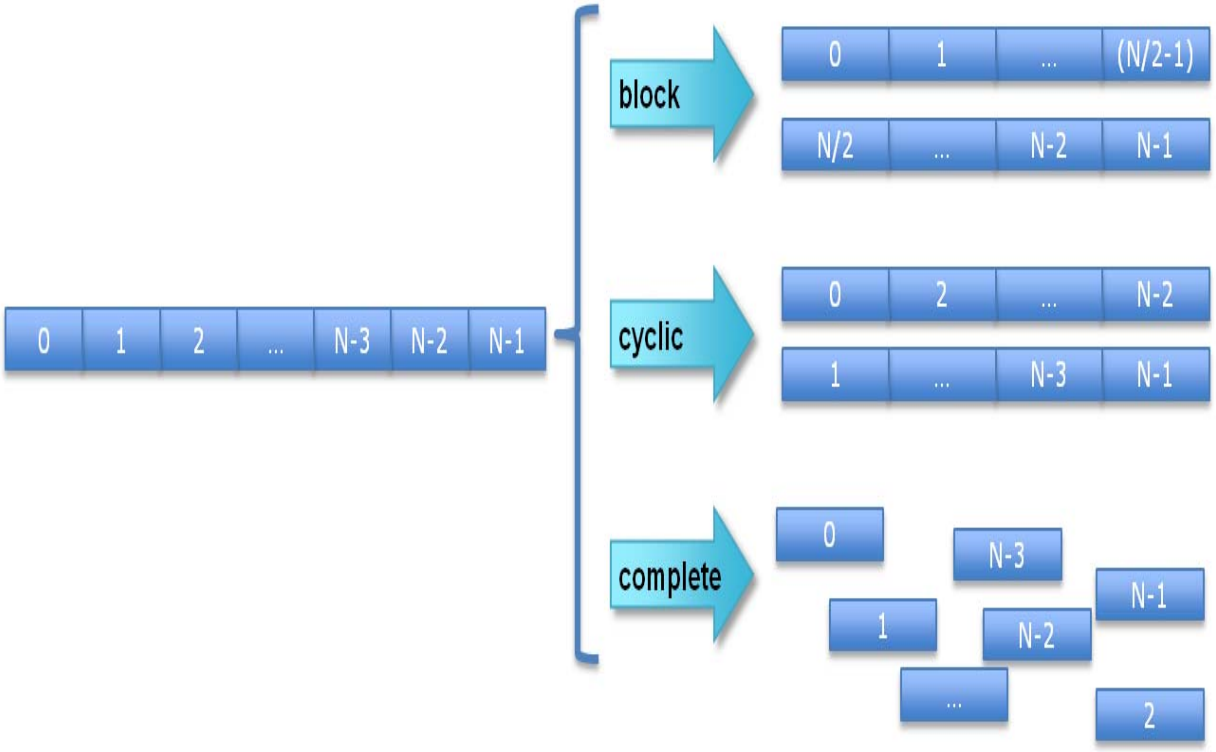

Figure 2-75: Array Partitioning

For block and cyclic partitioning the -factor option can be used to specify the number of array which are created. In Figure 2-75 a factor of 2 is used - the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final array has fewer elements.

When partitioning multi-dimensional arrays, the -dim option can be used to specify which dimension is partitioned. In this example,

```
void foo (...) {
   int  array1[L][M][N];
   ...
}
```

"array1" is split into three arrays, each of which has dimension 1 of size L/3.

```
set_directive_array_partition -type block -dim 1 -factor 3 foo array1
```

If zero is specified as the dimension (-dim 0) all dimensions are partitioned.

Array Reshaping

The command set_directive_array_reshape combines array partitioning with vertical mapping. This ultimately takes different elements from a dimension in the original array, and combines them into a single element in the reshaped array.

Given the following example:

```
void foo (...) {
   int  array1[N];
   int  array2[N];
   int  array3[N];
   ...
}
```

The following commands can be used to reshape arrays "array1", "array2" and "array3" into a three new arrays, using the default factor of 2 to create block, cyclic and complete types.

```
set_directive_array_reshape -type block -instance array4 foo array1
set_directive_array_reshape -type cyclic -instance array5 foo array2
set_directive_array_reshape -type complete -instance array6 foo array3
```

Figure 2-76 shows the result of the above commands.

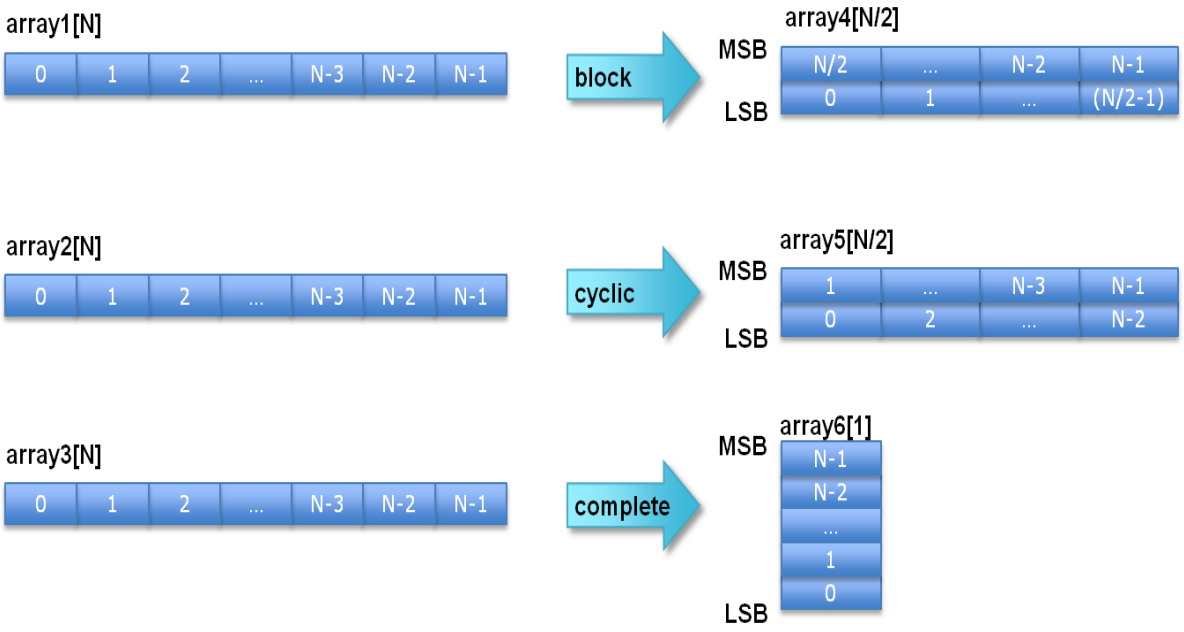

Figure 2-76: Array Reshaping

Array Streaming

By default all arrays are implemented as memory elements, unless complete partitioning reduces them to individual registers. This means all arrays will be assigned to a RAM resource and accessed with the data, address, chip and write enable signals specified on the RAM core in the technology library.

To use a FIFO instead of a RAM, the array must be specified as streaming. The following arrays are automatically specified as steaming:

- If an array on an interface is set as interface type ap_fifo it is automatically set as streaming.

All other array must be specified as streaming if a FIFO interface is required. This includes if streaming interface is required between function or loops when dataflow pipelining is specified.

An array can be specified to be streaming by

```
set_directive_array_stream  foo array1
```

The only options to the command are used when applying the command to arrays involved in dataflow channels. The -depth option overrides the default FIFO depth (the size of the largest array) and the -off option overrides the config_dataflow command: when the default channel is specified as a FIFO the set_directive_array_stream -off option can be used to prevent the array from streaming and ensure it uses a pingpong buffer.

Logic Structure Optimizations

The logic structures created by High-Level Synthesis are of the utmost importance. Once functions and loops have been optimized for concurrent operation, merged, inlined and flattened for minimum latency, pipelined for maximum throughput and array accesses have been analyzed to reduce bottlenecks, it is often ultimately the logic structures which dictate or limit the performance of a design and understanding how to improve them, the key to success.

A number of items dictate the type of logic structures implemented in a design:

- Clock Rate
- Target Device
- Operator Selection
- Controlling Hardware Resources
- Struct Packing
- Expression balancing
- Elaboration Effort

The impact of the clock, target device, state machine encoding and the reset are discussed in the "Design Optimization" chapter. These items should be reviewed and confirmed before applying any other optimization techniques discussed here.

Operator Selection

During synthesis High-Level Synthesis selects implementations for operators (+, -, *, /, %, etc.) from the device technology library. By default High-Level Synthesis chooses operators which are the best balance between timing and area. The config_bind can be used to influence which operators are used and to minimize the number of operators.

```
config_bind -effort [low | medium | high] -min_op <list> -reset
```

The config_bind command can only be issued inside an active solution. Once the command has been issued it will apply to all synthesis operations performed in the solution: if the solution is closed and re-opened the specified configuration will still apply to any new synthesis operations.

Any configurations applied with the config_bind command can be removed by using the -reset option or by using open_solution -reset to open the solution.

The default effort level for the binding operation is medium.

- **Low Effort**: Spend less timing sharing, run time is faster but the final RTL may be larger. Useful for cases where the designer knows there is little sharing possible or desirable and does not wish to waste CPU cycles exploring possibilities.
- **Medium Effort**: The default, where High-Level Synthesis tries to share operations but endeavors to finish in a reasonable time.
- **High Effort**: Try to maximize sharing and do not limit run time. High-Level Synthesis will keep trying until all possible combinations of sharing have been explored.

Effort levels impact every operator in the top-level function - bindings are set for the entire design.

The -min_op operation allows a particular operation to be minimized in the RTL. For example, if the design contains 12 multipliers, the following command would seek to minimize the number of multipliers in this design:

```
config_bind -min_op mul
```

Refer to the config_bind command in the "High-Level Synthesis Reference Guide" for a complete list of available operators.

Controlling Hardware Resources

The resources used to implement the RTL can be specified explicitly during synthesis or a general limit can be put on the resources synthesis is permitted to use. These techniques can be used to both improve timing (and hence latency and throughput) and area.

The resource used for a specific operation can be directly specified. Resources can be specified using the GUI (and as a pragma) and applied to any variable in a function using the set_directive_resource command.

When High-Level Synthesis reads the code shown in this example,

```
int foo (
  int a,
  int b
  ) {
  int c, d;
  c = a*b;
  d = a*c;
  return d;
}
```

the multiplications, used for variables "c" and "d", will be implemented in the internal database as standard "mul" (multiplier) operators. A complete list of operators is available in the "High-Level Synthesis Library Guide".

When synthesis is performed, High-Level Synthesis will use the timing constraints specified by the clock, the delays specified by the target device and any user constraints to determine which core is used to implement the operators: it could use the combinational core "multiplier" or it may decide to use a pipeline multiplier core such as "Mul2S". A complete list of available cores is provided in the "High-Level Synthesis Library Guide".

The RESOURCE directive can be used to explicitly specify which core should be used. The following command informs High-Level Synthesis to use a 2-stage pipelined multiplier for variable "c".

```
set_directive_resource -core Mul2S foo c
```

In addition to selecting specific operators to improve timing or area, the total number of operators used in the design can be limited to force operator sharing and improve area (often at the expense of timing or latency). For the same example code given above, the following command:

```
set_directive_allocation -limit 1 -type operation foo mul
```

limits the implementation to one mul operation (by default, there is no limit). This forces High-Level Synthesis to use a single multiplier for function "foo".

Struct Packing

Packing the members of a struct into a single wide-word can reduce the control overhead associated with each of the individual elements are result in both a smaller and faster design.

The PACK directive can be used to pack the three 8-bit char elements into a single wide-word element. In the new word, the first element of the struct will occupy the least significant bits and the last element, the most significant bits. If the struct contains arrays, the array with be reshaped with complete partitioning and packed with the other scalars.

Given a struct, “my_data”, used in function foo

```
typedef struct{
  unsigned char A;
  unsigned char B;
  unsigned char C;
}my_data;

void foo(my_data a_in[50], my_data b_out[50])
{

  int i;

  for(i=0; i < 50; i++){
    b_out[i].A = (a_in[i].A >> 1) + 10;
    b_out[i].B = (a_in[i].B >> 2);
    b_out[i].C = (a_in[i].C >> 3) + 100;
  }
```

```
}
```

The following commands will pack the members “a_in” into a new variable called “a_in” (it will use the same name if no –instance option is used) and pack the members of struct “b_in” into a new called “new_var”.

```
set_directive_data_pack foo a_in
set_directive_data_pack –instance new_var foo b_out
```

In both cases, the new variables will be 24-bits wide (three 8-bit char types).

Note: The maximum bit-width of any port or bus created by data packing is 8192 bits.

Expression Balancing

During synthesis a number of optimizations, such as strength reduction, bitwidth minimization etc. are performed automatically. Included in the list of automatic optimizations is expression balancing.

One of the optimizations which can be directly controlled is expression balancing. This optimization rearranges operators to construct a balanced tree and reduce latency. Expression balancing is on by default but may be disabled.

Software programmers often will write highly sequential code by using assignment operators (such as += and *=) for convenience. However, this sequential code may have an adverse effect on latency.

Let us look at the following example:

```
int foo_top (short a, short b, short c, short d)
{
  int i;
  int sum;

  sum = 0;
  sum += a;
  sum += b;
  sum += c;
  sum += d;
  return sum;

}
```

The code above must be executed sequentially. Suppose each addition requires one clock cycle, the complete computation for "sum" requires four clock cycles shown in Figure 2-77.

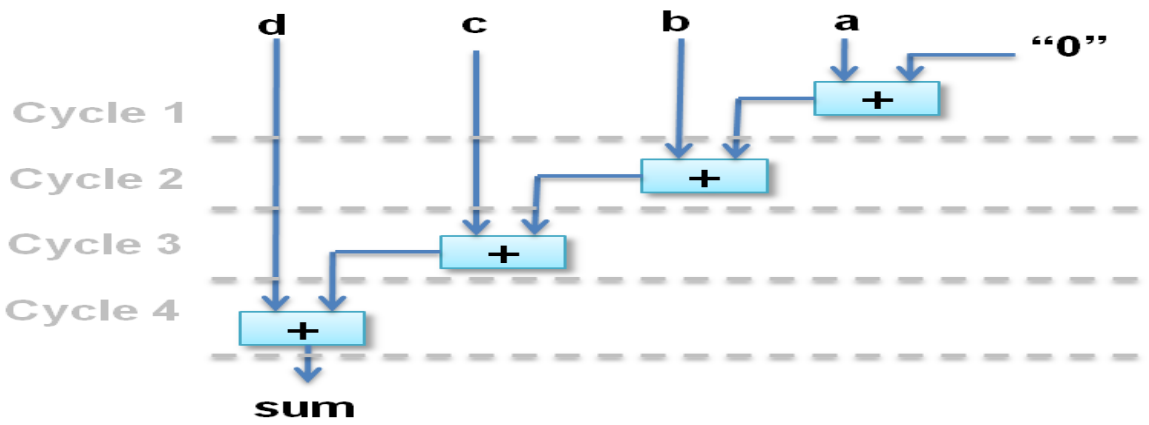

Figure 2-77: Adder Tree

However additions "(a+b)" and "(c+d)" can be executed in parallel allowing the latency to be reduced. After balancing the computation completes in two clock cycles as shown in Figure 2-78.

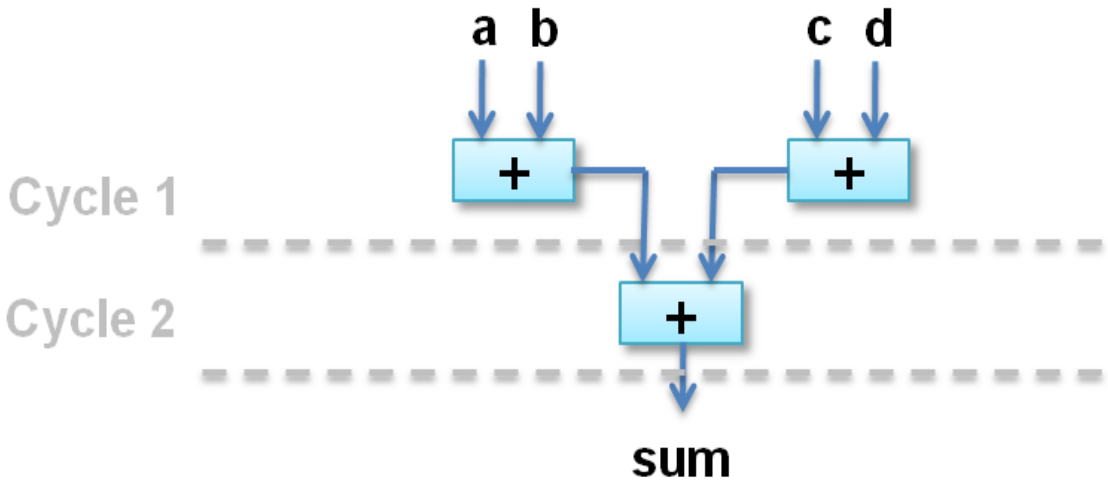

Figure 2-78: Adder Tree After Balancing

Expression balancing typically prohibits sharing and can result in increased area. In the example from Figure 2-77 a single adder can be used for the entire design, unless the design is pipelined, because only one adder is required in each clock cycle. In the example in Figure 2-78 a minimum of two adders are required (and three if the design is pipelined).

The following command turns off expression balancing in function "foo":

```
set_directive_expression_balance -off foo
```

Elaboration Effort

The first process performed on the input function(s) is elaboration. It is during elaboration that the functionality of the C/C++/SystemC is transformed into generic logic structures. The effort level used during elaboration has an impact on the starting point for synthesis.

By default, the effort level used during elaboration is std (standard). The standard effort level is typically enough for most design, because it provides the best balance between memory/CPU usage and optimization.

The effort level during elaboration can be set by using the -effort option.

```
elaboration -effort [low | std | high]
```

Low Effort Level

A low effort level is only recommended when the time taken to elaborate the design becomes excessively large. No optimizations will be performed on if-else or branch conditions which can result in larger area (due to less sharing).

Standard Effort Level

The default standard effort level seeks a balance between optimization and run time.

High Effort Level

When high effort level is used more optimization is performed on the initial database which helps both timing and area. Anything which increases the search space (a design with many mutually exclusive paths, many opportunities for sharing, a lack of design constraints etc.) will increase the run time.

Verification

Post-synthesis verification can be automated through the use of the cosim_design feature which can re-use the pre-synthesis test bench to seamlessly perform verification on the output RTL.

When synthesis completes High-Level Synthesis writes the output RTL to the "syn" directory as shown in Figure 2-79. These files can be used with an appropriately created RTL test bench to verify the design.

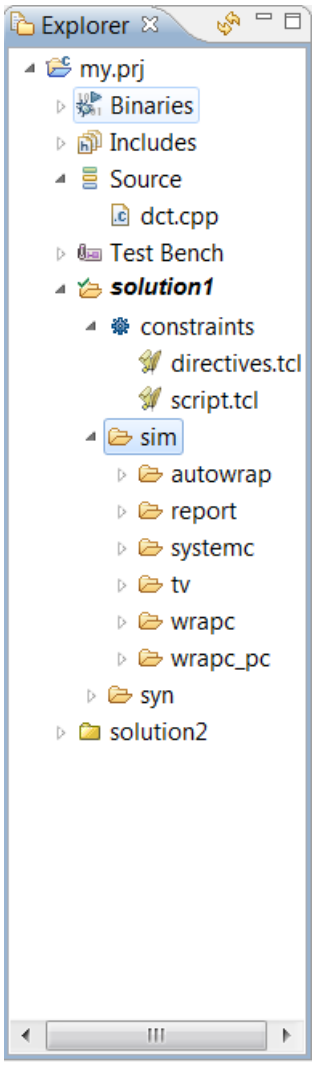

Figure 2-79: Output Directory Structure

However, High-Level Synthesis provides a much more productive method to verify the RTL design: cosim_design.

Automatic Verification of the RTL

The cosim_design feature can re-use the existing C-level test bench created for pre-synthesis verification and automatically verify the RTL using he built-in SystemC RTL simulator or a 3rd party HDL simulator.

The following is required in order to use the cosim_design feature successfully:

- The correct interface synthesis options must be selected.
- The test bench must be self-checking and return a value of 0.
- Any 3rd-party simulators must be available in the search path.

Interface Synthesis Requirements

In order to use the cosim_design feature to automatically verify the RTL design.

- The top-level function of C and C++ designs must be synthesized using an ap_ctrl_hs interface.
  - This interface creates a start, done and idle port on the design which are used to control when transactions begin and when to capture data at the end of a transaction.
- C and C++ designs must use one of the following interfaces on each output port. (These are the only interfaces which provide a data valid signal, required to capture the output data):
  - ap_vld
  - ap_ovld
  - ap_hs
  - ap_memory
  - ap_fifo
  - ap_bus

Since SystemC designs do not use interface synthesis, there are no such requirements for SystemC designs.

Unsupported Optimizations

The automatic RTL verification does not support cases where multiple transformations have been performed upon arrays or arrays within structs on the interface.

In order for automatic verification to be performed, arrays on the function interface, or array inside structs on the function interface, can use any of the following optimizations, but not two or more:

- Reshape.
- Partition.
- Data Pack on structs elements.

Test Bench Requirements

To verify the RTL design produces the same results as the original C code, the test bench used to execute the verification should be self-checking. The important features of a self-checking test bench are discussed in the following example:

```
int main () {
  int ret=0;
  …
  // Execute (DUT) Function
  …

  // Write the output results to a file
  …

  // Check the results
  ret = system("diff --brief  -w output.dat output.golden.dat");

  if (ret != 0) {
        printf("Test failed  !!!\n");
        ret=1;
  } else {
        printf("Test passed !\n");
  }
  …
  return ret;
}
```

- Write the output from the function to a file.
- Compare the results to some existing know good (or golden) results.
- If the results are correct, return the value 0.
- If the results are incorrect, return a non-zero value.
  - Any value can be returned. A sophisticated test bench may return different values depending on the type of difference/failure.

A test bench such as the one shown above provides a substantial productivity improvement by automatically checking the results, freeing the user from manually verifying them.

If a zero is returned by the top-level test bench function main(), High-Level Synthesis will issue a message stating the verification was successful.

```
@I [SIM-1] *** cosim_design finished: PASS ***
```

Note: If the test bench returns a value of zero, but does not self-check the RTL results and confirm the results are indeed correct, High-Level Synthesis will still issue message SIM-1 (as above) indicating the simulation test passed: when no results have actually been checked.

Ensure the test bench checks the results against the expected behavior.

If any non-zero value is returned, High-Level Synthesis will issue two messages: one stating the value returned and one stating the RTL verification failed.

Note: A return of "20" also means there was no return value in the test bench: the test bench should be enhanced to self-check the results and return a value of zero if they are correct.

Debugging a Simulation Mismatch

If the test bench is self-checking and shows the results from the RTL to be different from the C code, the following methodology can be used to debug the differences and confirm this is a bug in the RTL:

1. Confirm the result from the C validation step was not created by from a double or float type. When comparing the results of double or float types, the test bench must be smart enough to compare in ranges and not in absolute values, since associatively optimizations (the order of the operations) can vary the results depending on the level of optimization applied in the C compilation and synthesis.
2. If required, modify the test bench to print at which sample/cycle the difference is first observed.
3. When executing the simulation, as shown below, select the Dump Trace option to create a VCD and view the output waveforms in a 3rd-party tool which can open VCD files (e.g. ModelSim, Verdi, etc.).
4. Try to match up common points in the C and RTL and using the debugger in the CDT and the RTL VCD file, determine when and where the two representations diverge.

The debug steps may also include adding printf statements or writing specific results to a file in the C source but the above is the basic methodology for debugging differences.

RTL simulator support

With the above requirements in place, cosim_design can verify the RTL design using any of the valid language and simulator combinations shown in Table 2-11.

Table 2-11: Cosim_design Simulation Support

Simulator	OSCI	ModelSim	VCS
SystemC	Supported	Not Supported	Not Supported
Verilog	Not Supported	Supported	Supported
VHDL	Not Supported	Supported	Supported

When verifying the SystemC RTL output, High-Level Synthesis uses the built-in SystemC kernel to verify the RTL. This does not require a license, uses the same version of SystemC used in synthesis and means the RTL design can always be verified using High-Level Synthesis.

To verify one of the RTL HDL designs (verilog or VHDL) any of the 3rd-party simulators shown in Table 2-12 may be used.

For the 3rd party simulators, the executable must be available in the OS search path and the simulator must have a C co-simulation license, since the original C test bench must also be simulated with the design.

RTL Verification

The simulation can be launched from the GUI using the simulation toolbar button.

Figure 2-80: **Tool Bar**

This in turn opens the simulation wizard window (Figure 2-81).

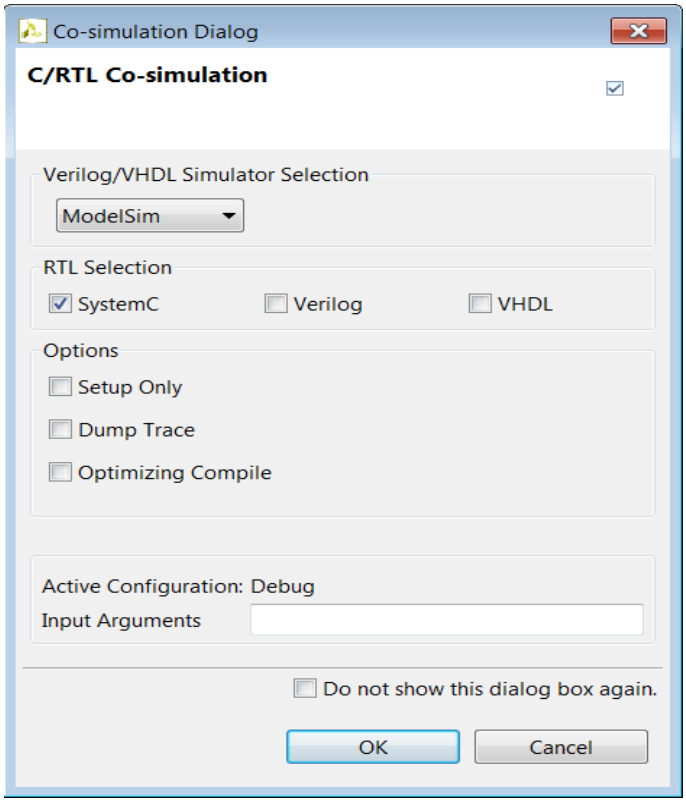

Figure 2-81: High-Level Synthesis Simulation Wizard

The wizard presents the available RTL languages to simulate (any or all can be simulated). To simulate using a specific language, use the drop-down menu to select a setting other than "Skip": the drop-down menu will list the simulators supported for that language.

Alternatively the simulator can be run using the High-Level Synthesis command line interface using the cosim_design Tcl command:

```
# Simulate VHDL RTL using the ModelSim simulator
cosim_design -tool modelsim -rtl vhdl

#Simulate systemc RTL using the OSCI simulator
cosim_design -rtl systemc
```

Once the verification has been executed, the sim directory shown in Figure 2-79 will be populated by the simulation files and the adapters used within test bench: these are not intended for user review but they are not encoded or protected.

If the Setup Only option is selected, High-Level Synthesis will create the scripts, adapters and wrappers to verify the design but will not execute the simulator. The Dump Trace option

causes a VCD trace file to be written for each function in the design: these files are written to the appropriate HDL sub-directory of the "sim" directory shown in Figure 2-79.

The Optimizing Compiler option will result in High-Level Synthesis using optimized options to compile the C test bench and SystemC adapters: this will result in longer compile time but will improve the run time performance.

The Input Arguments allows the specification of any arguments required by the test bench. The active compile configuration is shown as part of this dialog box.

Exporting the RTL Design

The final step in the Vivado HLS flow is to export the RTL design as a block of Intellectual Property (IP) which can be used by other tools in the Xilinx flow. The RTL design can be exported as three types IP Block:

- IP-XACT formatted IP for use with Vivado
- A System Generator IP block
- Pcore formatted IP for use with EDK

Each of these export formats has some restrictions related to the device technology used to implement the design and the version of Vivado HLS being used. Table 2-12 shows a summary of which technologies can be exported in each version of Vivado HLS. Refer to section Vivado HLS Licensed Technologies to understand the difference between Vivado HLS (System Edition or SE) and Vivado HLS (standalone) versions.

Table 2-12: Device Support for RTL Export Flows

	Vivado HLS (SE)	Vivado HLS (Standalone)
IP-XACT	7-Series	7-Series
System Generator	7-Series	7-Series
EDK Pcore	Not Supported	7-Series, Spartan-3, Spartan-6, Virtex-4, Virtex-5, and Virtex-6

Generally, the output package will contain both Verilog and VHDL RTL, however, if the design uses any bus interfaces (AXI4. PLB, FSL interfaces etc) only Verilog RTL will be output in the package.

Bus Interfaces

Since buses interfaces allow the IP block to be more easily connected to other blocks in a system, it is common when exporting IP blocks to make extensive use bus interfaces (AXI4, PLB, FSL etc.).

Before proceeding to export the RTL, refer to the section Specifying Bus Interfaces in the Interface Management chapter to ensure you have correctly specified any bus interfaces.

As noted above, the use of bus interfaces means only Verilog RTL will be output in the exported IP.

RTL Synthesis

When Vivado HLS reports on the results of synthesis, it provides estimations of the results expected after RTL synthesis: the expected clock frequency, the expected number of registers, LUTs and BRAMs etc. These results are estimations because Vivado HLS cannot know what exact optimizations down-stream RTL synthesis will perform or what will be the actual routing delays, and hence final timing, after place and route.

Before exporting a design, you have the opportunity to execute logic synthesis, for either the Verilog or VHDL version of the design, using the evaluate option shown in Figure 2-82.

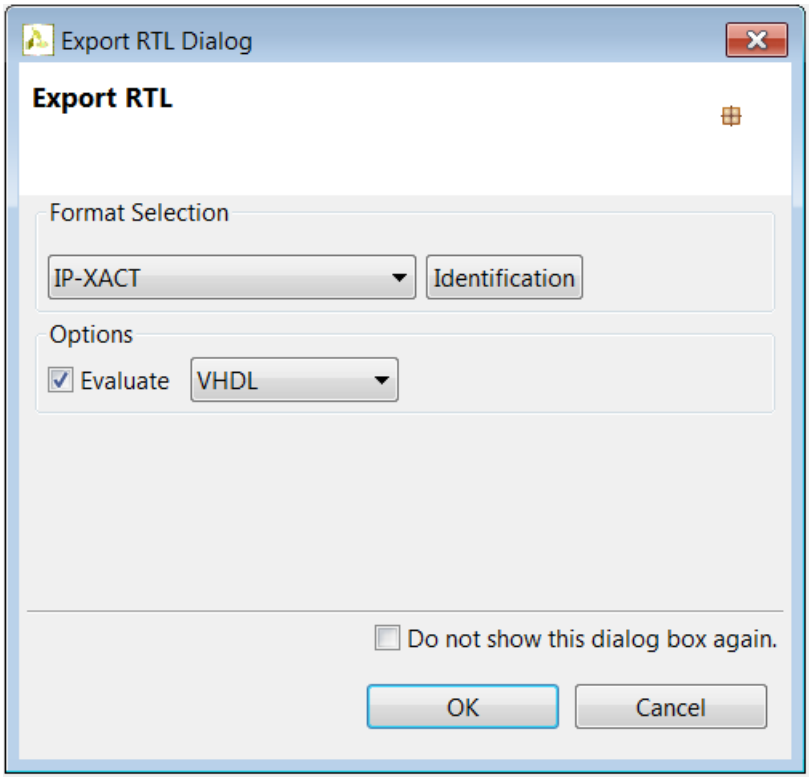

Figure 2-82: Export RTL Dialog Box

Logic synthesis will automatically be performed by the appropriate Xilinx RTL synthesis product:

- 7-Series devices will use Vivado RTL Synthesis.
- Zynq and Non-7-Series devices will use ISE.

Note: To execute RTL synthesis, the logic synthesis binary executable must be available in the system search path. Refer to the Vivado HLS Installation Guide.

This evaluation option simply allows the HLS estimations to be confirmed from within the Vivado HLS environment, before the IP is sent to the next stage of the design flow. The

results of RTL synthesis are stored in the directory `<Project_Directory>/<Solution_Name>/impl/<HDL_Version>` and are not part of the exported IP.

Keep in mind, when this RTL IP block is included in a larger RTL design and RTL synthesis re-executed, these results may change slightly: the evaluate option is only meant to provide quick confirmation that the final results will likely be close to the Vivado HLS estimates.

Package Identification

The IP-XACT and Pcore formats support identification tags embedded in the exported package. These fields are optional but will use the following default strings if none are applied (the pcore format includes only the version identification string).

- version: 1.00.a
- library: hls
- vendor: xilinx.com
- description: An IP generated by Vivado HLS

Tcl command

RTL export is supported by the Tcl command `export_design`. This command can only be issued after the `csynth_design` command but can be executed multiple times with different options.

Refer to the Vivado HLS Reference Guide or the GUI menu Help > Man Page for more details on this command.

Exporting in IP-XACT Format

Upon completion of synthesis and RTL verification, open the `Export RTL` dialog box by selecting the `Solution` > `Export RTL` from the main menu, or by right-clicking on the desired solution in the Explorer pane, or by clicking on the `Export RTL` toolbar icon as shown in Figure 2-83.

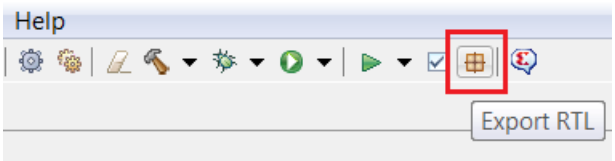

Figure 2-83: Export RTL Tool Bar Button

Select IP-XACT in the Format Selection section (Figure 2-82). This is the default choice if it has never been changed for a given solution.

Identifying information fields for the IP-XACT package may be customized in the IP Identification dialog box. Default values are populated for all fields.

If post-place-and-route resource and timing statistic for the IP block are desired then select the Evaluate option and the desired RTL language.

Pressing OK will generate the IP-XACT package. This package will be written to directory <Project_Directory>/<Solution_Name>/impl/ip. This ip directrory will contain a .zip archive: this is the IP-XACT package.

If the Evaluate option was selected, logic synthesis will be executed and the final timing and resources reported. Refer to the RTL synthesis section above for details on RTL synthesis.

Importing IP-XACT package into Vivado

A Vivado HLS generated IP-XACT package may be imported into the IP Catalog of Vivado a project by following these steps in the Vivado GUI (see the Vivado documentation for other methods of importing IP):

1. Open the project in the Vivado GUI.
2. Click on Project Manager > IP Catalog in the Flow Navigator pane (far left in default layout)
3. In the IP Catalog tab (upper right pane in default layout) click on the Add IP icon (lower left; yellow symbol w/ green '+'). This will bring up the Add IP dialog.
4. If no Repository Path is setup or if a different one is preferred, select or create it
5. Click on the browse icon (ellipsis ...) which will bring up the Select IP File dialog
6. Browse to and select to the .zip archive for the HLS generated package. Click OK for this dialog and the previous one.
7. After processing the IP should appear under the heading "VIVADO HLS IP"
8. Add the IP to the as with any other Xilinx provided IP from this point

The repository directory to which user generated IP is imported may be added to the IP Catalog for all future projects by selecting the Tools > Options menu item, selecting the General category, scrolling down to the IP Catalog item and Add Directories.

Exporting in Pcore Format

Upon completion of synthesis and RTL verification, open the Export RTL dialog box by selecting the Solution > Export RTL from the main menu, or by right-clicking on the

desired solution in the Explorer pane, or by clicking on the `Export RTL` toolbar icon as shown in Figure 2-83.

Select `Pcore for EDK` in the `Format Selection` section, as shown in Figure 2-84. The version information field for the Pcore package may be customized in the IP `Identification` dialog box. A default value of `1.00.a` is used if none is specified.

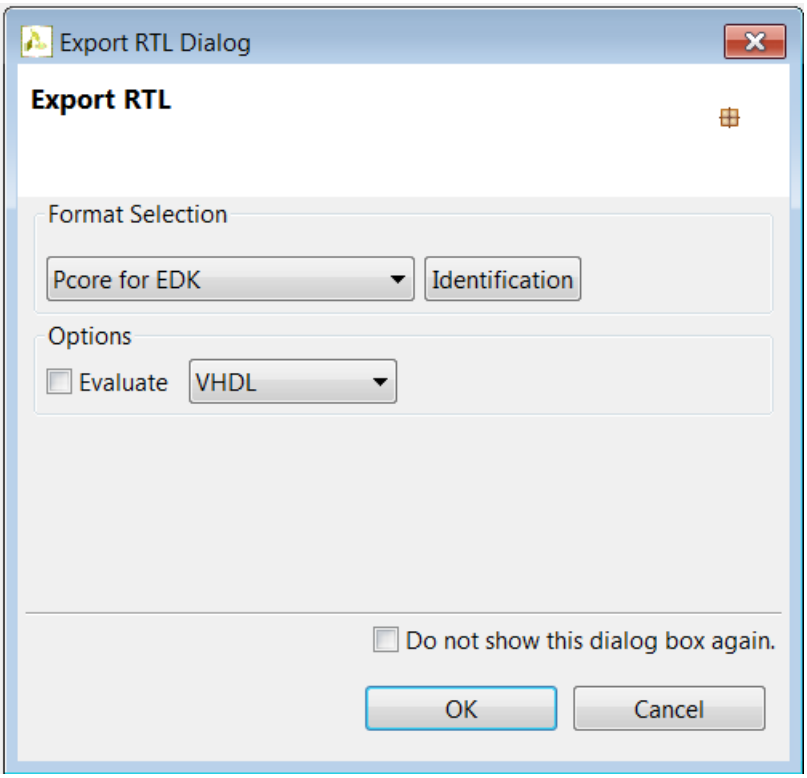

Figure 2-84: Export RTL as Pcore

If post-place-and-route resource and timing statistic for the IP block are desired then select the `Evaluate` option and select the desired RTL language.

Pressing `OK` will generate the `Pcore` package. This package will be written to directory `<Project_Directory>/<Solution_Name>/impl/pcores`. This `pcores` directory will contain a sub-directory named `<top_level_design_name>_top_v<Version_String>`: this is the `Pcore` package.

Within the `Pcore` package the directory `include` will contain the software files for any slave interface in the design: AXI4 Lite Slave or PLB 4.6 (slave) interfaces. Refer to the section Specifying Bus Interfaces in the Interface Management chapter for details on how these files are used.

If the `Evaluate` option was selected, logic synthesis will be executed and the final timing and resources reported. Refer to the RTL synthesis section above for details on RTL synthesis.

Importing a Pcore package into the EDK environment

A Vivado HLS generated Pcore package may be imported into the EDK environment by simply copying the contents of the pcores directory to the pcores directory in the EDK project.

1. Copy the directory `<Project_Directory>/<Solution_Name>/impl/pcores/*` to the directory `<EDK_Project>/pcores`.
2. From within the EDK project, the IP block will be listed under `Project Local PCores`.
3. If the IP block is not listed under `Project Local PCores`, use `Project > Rescan Local Repository` to manually update the local Pcore information.

Exporting To System Generator

Any IP block imported into System Generator must have a clock-enable port. Vivado HLS will check to ensure this condition is satisfied before exporting any design to the System Generator environment. Vivado HLS will issue an error message, as shown below, if the design does not have a clock enable port.

IMPORTANT: *@E [IMPL-64] Clock enable port is required for exporting the design to System Generator. Please use 'config_interface -clock_enable' to generate the port.*

Note: Vivado HLS must be configured to add a clock-enable port to the design before C synthesis is performed.

As explained in the error message above, a clock-enable port can be added to any Vivado HLS design by using the `config_interface` option `clock_enable`.

This interface configuration can be performed using the GUI or by Tcl command:

A. In the GUI, select the desired solution to ensure it is the current active solution. Open the interface configuration setting using the menus `Solution > Solution Settings > General > Add > config_interface` and select the `clock_enable` option.

OR

B. Add the following Tcl command `config_interface -clock_enable` before the `csynth_design` command in your Tcl script

Port Optimizations

If any top-level function arguments are transformed during the synthesis process into a composite port, the type information for that port cannot be determined and included in the System Generator IP block.

The implication for this limitation is that any design which uses the reshape, mapping or data packing optimization on ports must have the port type information, for these composite ports, manually specified in System Generator.

To manually specify the type information in System Generator, you should know how the composite ports were created and then use slice and reinterpretation blocks inside System Generator when connecting the Vivado HLS block to other blocks in the system.

For example,

- If three 8-bit in-out ports R, G and B are packed into a 24-bit input port (RGB_in) and a 24-bit output port (RGB_out) ports.

Once the IP block has been included in System Generator:

- The 24-bit input port (RGB_in) would need to be driven by a System Generator block which correctly groups three 8-bit input signals (Rin, Gin and Bin) into a 24-bit input bus.
- The 24-bit output bus (RGB_out) would need to be correctly split into three 8-bit signals (Rout, Bout and Gout).

Refer to the System Generator documentation for details on how to use the slice and reinterpretation blocks for connecting to composite type ports.

Exporting the RTL

Upon completion of synthesis and RTL verification, open the `Export RTL` dialog box by selecting the `Solution` > `Export RTL` from the main menu, or by right-clicking on the desired solution in the Explorer pane, or by clicking on the `Export RTL` toolbar icon as shown in Figure 2-83.

Select `System Generator For DSP` in the `Format Selection` section, as shown in Figure 2-85.

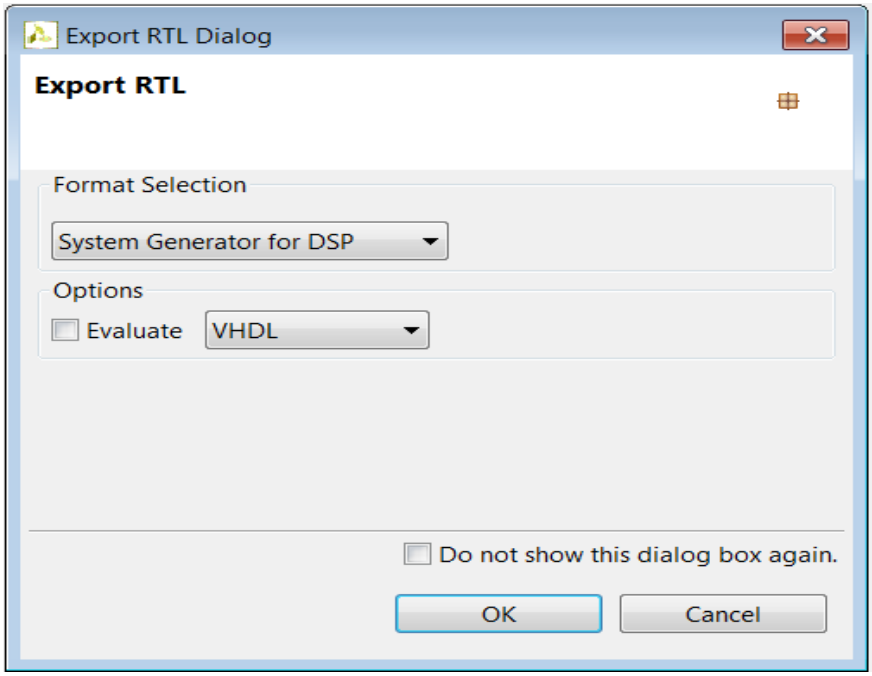

Figure 2-85: Export RTL to System Generator

If post-place-and-route resource and timing statistic for the IP block are desired then select the `Evaluate` option and select the desired RTL language.

Pressing `OK` will generate the `System Generator` package. This package will be written to directory `<Project_Directory>/<Solution_Name>/impl/sysgen`. And contains everything need to import the design to System Generator.

If the `Evaluate` option was selected, logic synthesis will be executed and the final timing and resources reported. Refer to the RTL synthesis section above for details on RTL synthesis.

Importing the RTL into System Generator

A Vivado HLS generated System Generator package may be imported into System Generator using the following steps:

1. Inside the System Generator design, right-click and use option XilinxBlockAdd to instantiate new block.
2. Scroll down the list in dialog box and select Vivado HLS.
3. Double-click on the newly instantiated Vivado HLS block to open the Block Parameters dialog box.

4. Browse to the solution directory where the Vivado HLS block was exported. Using the example above, `<Project_Directory>/<Solution_Name>/impl/sysgen`, this would mean browse to directory `<Project_Directory>/<Solution_Name>` and select `apply`.

High-Level Synthesis Operator and Core Guide

High-Level Synthesis transforms a C, C++ or SystemC design specification into a Register Transfer Level (RTL) implementation which in turn can be synthesized into a Xilinx Field Programmable Gate Array (FPGA).

To perform this task High-Level Synthesis, does the following:

- First elaborate the C, C++ or SystemC source code into an internal database containing operators.
    - The operators represent operations in the C code such as additions, multiplications, array reads and writes etc.
- During synthesis, High-Level Synthesis maps the operators to cores from the High-Level Synthesis library.
    - Cores are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAMs)
    - A separate library is provided for each Xilinx technology (Spartan®-6, Virtex®-7, and other Xilinx® devices)

This document provides details on the operators supported by High-Level Synthesis and the accompanying libraries.

Synthesis Overview

When High-level Synthesis (HLS) is performed to transform C source code into a Register Transfer Level (RTL) description, the source code is elaborated into an internal database which contains operators.

The operators are explained in this *High-Level Synthesis Operator and Core Guide*, but basically consist of all the operations which can occur in the C source code, such as: additions, shifts, multiplications, bit-slicing, array accesses, and so forth.

High-Level Synthesis uses this internal database when it synthesizes the design. Synthesis is a two-step process consisting of *scheduling* and *binding*.

About Scheduling

Scheduling is where High-Level Synthesis determines in which cycles an operation is to occur.

If, for example, two addition operations are scheduled in the same clock cycle they cannot use the same hardware adder; however, if they are scheduled in different clock cycles, they could use the same adder and save resources. In the absence of any constraints or directives High-Level Synthesis first tries to schedule the design to achieve the minimum possible latency: this could require scheduling two additions in the same clock cycle.

About Binding

When scheduling completes, binding is the process where the scheduled operations are bound to specific hardware implementations (or cores) from the technology library.

For example, a multiplication operation in the source code could be implemented by a standard combinational multiplier, while another multiplication could be implemented using a pipelined multiplier: the fact that a pipelined multiplier requires two stages would have been considered during scheduling.

In High-Level Synthesis the effects of binding (knowledge of the specific cores that are used to implement operations) are considered during the scheduling process. Figure 1, page 188 shows the process of scheduling and binding. This prevents decisions made during binding from requiring that the design be re-scheduled, preventing endless iterations.

The types of cores available during the binding process depend on the selected device. High-Level Synthesis provides a unique library for each Xilinx device and hence cores from different libraries have different delays and timing. The delays associated with each core affect which cores can be scheduled in a single clock cycle.

The created schedule depends upon to what cores the operators are bound. As such, the scheduling process takes into consideration the effects binding has on the design.

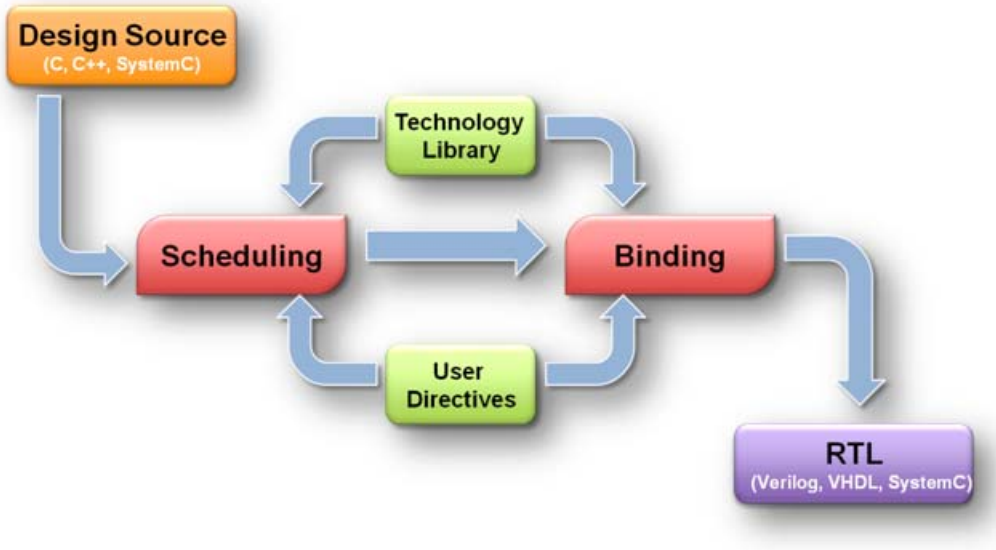

Figure 3: **Synthesis Process**

In addition to device selection, High-Level Synthesis provides a number of commands and directives that let you control the scheduling and binding process.

The process of synthesis, although straightforward to describe, is complicated in implementation, taking account of a number of factors when creating a designs, such as the:

- Operations in the code
- Design schedule
- Timing delays of the cores
- User constraints and directives
- Binding process

Understanding which cores are available for the RTL implementation is often crucial to achieving a high performance design. The following subsections describes the High-Level Synthesis operations and cores.

Understanding Operators, Cores & Directives

During High-Level Synthesis (HLS) the operations in the C, C++ or SystemC source code are identified and represented in the internal database. There are no commands to list or access the internal database; the operations can be seen in the Design Viewer that is available in the High-Level Synthesis GUI. Figure 2, page 189 shows an example from the Design Viewer.

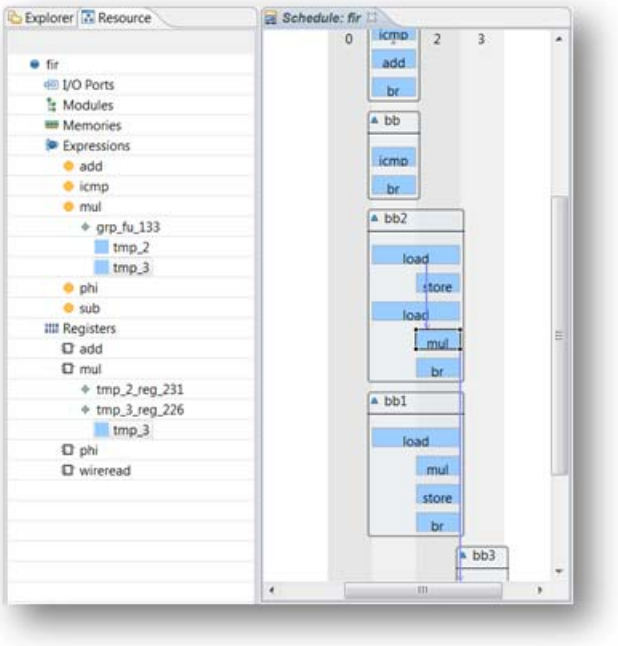

Figure 4: **Operations in the Design Viewer**

In Figure 2,a number of operations can be seen. For example, there are additions represented by an `add` operation, multiplications represented by a `mul` operation, array reads (`load` operation), comparisons (`cmp` operation) and breaks represented by a `br` operation.

Figure 2-85 also shows how the Design Viewer can show the result of operator binding. When the `mul` operation in block `bb2` is selected in the schedule viewer window (right side of the figure) it automatically cross-highlights in the resource viewer window (left side of the figure) showing this `mul` operation is bound to hardware multiplier resource and instance `grp_fu_133`. Instance `grp_fu_133` (the same instance name is used in the RTL) also shows it is used for a second multiplier operation: there are two operations inside `grp_fu_133`, `tmp2` and `tmp3`, indicating this single hardware instance is being used for two `mul` operations.

- Operations such as multiplications are implemented by a specific hardware multiplier in the RTL design using a specific core. Not all operations map to cores.
- Operations such as the break from a loop or switch statement indicate a control flow action and are implemented using logic rather than a core from the library. Operations such as these cannot be controlled by user directives.

Controlling Operators & Cores

High-Level Synthesis provides the following ways to control the use of operations and cores:

- Directing the allocation process for operations
- Directing the specific resources used for operations
- Scheduling efforts
- Control the binding process of operators to cores
- Listing details on the cores

Limiting Operators

High-Level Synthesis lets you limit how many operators are used in a design. For example, if a design called `foo` has 317 multiplications but the FPGA only has 256 multipliers, the following allocation command can be used to direct High-Level Synthesis to only create a design schedule with maximum of 256 multiplication (`mul`) operators:

```
set_directive_allocation -limit 256 -type operation foo mul
```

The `-type` option has specified `operation` because the allocation directive can be used in the same manner to limit the number of instances of cores and specific sub-functions.

Explicitly limiting the number of operators to reduce area might be required in some cases because the default operation of High-Level Synthesis is to first maximize performance:

- By default, in the absence of any constraints or directives, High-Level Synthesis tries to create a design with the lowest latency (the fewest number of cycles from input to output).
- When directives are applied to specify a maximum or minimum latency, High-Level Synthesis seeks to satisfy these latency constraints.
- When directives are applied for pipelining High-Level Synthesis first seeks to satisfy these constraints, and then minimizes or satisfies any latency constraints.

Minimizing latency can often mean using more cores in the final design, for example, using two adders in the same clock cycle rather than taking two clock cycles and sharing the same adder.

Limiting the number of operators ensures fewer cores are used in the final design and might force an increase in latency.

Table 1, page 194 lists all the operations which can be controlled using the `set_directive_allocation` command.

Controlling Resources

The `set_directive_resource` command specifies which core to use for an operation. This directive ensures the exact core is known during the scheduling process: no effort on binding is performed because this directive explicitly specifies the binding.

Listing the details on the technology library shows which operations can be implemented with each core (resource), and is explained Core Details, page 193.

The `set_directive_resource` directive is most typically used to specify which memory element is to be used to implement an array.

High-Level Synthesis can determine through analysis which cores can and should be used for each operation.

KEY CONCEPT: *For arrays, a specific memory from the technology library should be specified for each array: if none is specified High-Level Synthesis will automatically select a memory, single or dual-port, which provides the highest throughput and lowest latency.*

The resource directives uses the assigned variable as the target for the resource. Given code `Result=A*B` in function `foo`, this example specifies the multiplication be implemented with two-stage, pipelined multiplier core, `Mul2S`.

```
set_directive_resource -core Mul2S foo Result
```

If the variable is used with multiple operators, the code must be modified to ensure there is a single variable for each operator. For example:

```
Result = (A*B) + C;
```

Should be changed to:

```
Result_tmp = (A*B);
Result = Result_tmp + C;
```

And the directive specified on `Result_tmp` to control the multiplier resource or on `Result` to control the adder resource.

Controlling Schedule

The `config_schedule` command controls the effort during scheduling and can identify the exact path when the design fails to satisfy the constraints.

Scheduling Effort

The following general information can be used for all effort level controls in High-Level Synthesis, which are: **high**, **low**, and **medium**.

With an effort level set to **high**, High-Level Synthesis uses additional CPU cycles and memory, even after satisfying the constraints, to determine if it can create an even smaller or faster design. This exploration might, or might not, result in a better quality design but it does take more time and memory to complete. For designs which are just failing to meet their goals or for designs where many different optimization combinations are possible, this could be a useful strategy.

For some designs, the nature of the code does not allow much optimization. For example, if the code says move data from one variable to the next, there are no other actions to be done, and the code just needs to be implemented: the High-Level Synthesis software, however, does not determine that spending time doing exploration will not result in much improvement, and it spends time researching more possibilities. For designs such as this, an effort level set to **low** will most likely come to the same result much faster.

In summary, it is a better practice to leave the effort level at the default **medium** setting, however:

TIP: If the design has little room for various combinations of operators and cores *and it is running a long time*, using a low effort may give the same results much faster.

TIP: If the design *is just failing to meet the require performance in area or timing*, it is worth using a high effort to see what is possible with further exploration and trials.

Critical Path Analysis

The `config_schedule` command also has a `-verbose` option. When you specify this option High-Level Synthesis prints out the full critical path when scheduling is unable to meet the constraints.

Controlling Binding

The `config_bind` command provides control over the binding process. The command lets you direct how much effort is spent when binding cores to operators and enables direct control for operator minimization in area sensitive designs.

Binding Effort

The same general guidelines for scheduling effort also apply to binding. In this case, designs with operations for which there are a large number of possible cores will benefit more from higher efforts than design were there are few choice.

The `list_core` command, described in Core Details, can be used to determine the number of possible cores with which each operator can be implemented.

Minimizing operators

You can use the `config_bind` command to force more sharing on the design.

- The `-min_op` option to the `config_bind` command instructs High-Level Synthesis to create a design with the minimum number of specified operators.

For example, the following instructs High-Level Synthesis to create a design with the minimum number of `add` operators.

```
config_bind -min_op add
```

Because this command affects the binding process it only has an impact when operators are scheduled in the different clock cycles. If the operators are scheduled in the same clock cycle to satisfy other constraints (latency and/or throughput) the `config_bind` command has no effect on these operators. The allocation directive which impacts the result of scheduling should be used to first limit the number of operators.

This command option is typically used to override any consideration of MUXing costs. When operations are shared onto the same core, the additional MUXes, which are implemented in LUTs, can have a significant impact on both timing and area. High-Level Synthesis typically defaults to being conservative with MUXes if there is a potential that it will violate timing.

Core Details

The `list_core` command is used to obtain details on the cores available in the library.

To use the `list_core` a device must be selected using the `set_part` command. If no device has been selected, the command will have no effect.

- The `-operation` option of the list core command lists all the cores in the library that can be implemented with the specified operation.

Table 1, page 194 gives a complete list of the operations which can be queried. Using the command without any option lists all the cores available for the target device..

- The `-type` option can be used to further refine the cores by category:

- **Function Units**: Cores that implement standard RTL operations (such as add, multiply, compare)
- **Storage** - Cores that implement storage elements such as registers or memories.
- **Adapter** - Cores that implement interfaces used to connect the top-level design when IP is generated. These interfaces are implemented in the RTL wrapper used in the IP generation flow, such as the Embedded Development Kit (EDK).
- **IP Blocks** -Any IP cores added by the user.
- **Connectors** - Cores used to implement connectivity within the design. This includes direct connections and streaming storage elements.

Tables in High-Level Synthesis Cores, page 195 list the standard cores used in Xilinx devices.

High-Level Synthesis Operators

Table 1 lists the operators used by High-Level Synthesis.

The columns in the table indicate whether the operator is:

- Available for viewing in the Design Viewer
- Can be controlled by the `set_directive_allocation`, and `set_directive_resource` directives or the `config_bind` command
- If the associated cores can be listed from the library

Table 1: **High-Level Synthesis Operators**

Operator	Description	Design Viewer	Controlled by Directives	Library Core listed
add	Addition	X	X	X
ashr	Arithmetic Shift-Right	X	X	X
br	Break operation	X		
fiforead	FIFO Read	X		X
fifowrite	FIFO Write	X		X
fifonbread	Non-Blocking FIFO Read	X		X
fifonbwrite	Non-Blocking FIFO Write	X		X
icmp	Integer Compare	X	X	X
load	Memory Read	X		X
lshr	Logical Shift-Right	X	X	X

Table 1: **High-Level Synthesis Operators *(Cont'd)***

Operator	Description	Design Viewer	Controlled by Directives	Library Core listed
mul	Multiplication	X	X	X
mux	Multiplexor	X		X
phi	Multiplexor	X		
sdiv	Signed Divider		X	
shl	Shift-Left	X	X	X
srem	Signed Remainder	X		X
store	Memory Write	X		X
sub	Subtraction	X	X	X
udiv	Unsigned Division	X	X	X
urem	Unsigned Remainder	X		X
srem	Signed Remainder	X		X
wireread	IO read operation	X		
wirewrite	IO write operation	X		

High-Level Synthesis Cores

The High-Level Synthesis cores can be listed in the following categories (also used in the `list_core` command):

- Functional Unit Cores
- Storage Cores
- Connector Cores
- Adapter Cores
- Floating Point Cores
- IP Blocks

IMPORTANT: IP blocks are blocks added to the library by the user; consequently, they are not listed in this document.

The cores are explained in the Table 2 through Table 6, page 198 in the following subsections.

Functional Unit Cores

Table 2 lists the cores that implement standard RTL logic operations (such as add, multiply, and compare).

Table 2: **Functional Cores**

Core	Description
AddSub	This core is used to implement both adders and subtractors.
AddSubnS	An N-stage pipelined adder or subtractor. High-Level Synthesis will automatically determine how many pipeline stages are required.
Cmp	Comparator.
Div	Divider.
Mul	Combinational multiplier.
Mul2S	2-stage pipelined multiplier.
Mul3S	3-stage pipelined multiplier.
Mul4S	4-stage pipelined multiplier.
Mul5S	5-stage pipelined multiplier.
Mul6S	6-stage pipelined multiplier.
MulnS	N-stage pipelined multiplier. High-Level Synthesis will automatically determine how many pipeline stages are required.
Sel	Generic selection operator, typically implemented as a mux.

Storage Cores

Table 3 lists the cores that implement storage elements such as registers or memories.

Table 3: **Storage Cores**

Core	Description
FIFO	A FIFO. High-Level Synthesis will determine whether to implement this in the RTL with a BRAM or as distributed RAM.
FIFO_ BRAM	A FIFO implemented with a BRAM.
FIFO_LUTRAM	A FIFO implemented as distributed RAM.
FIFO_SRL	A FIFO implemented as with an SRL.
RAM_1P	A single-port RAM. High-Level Synthesis will determine whether to implement this in the RTL with a BRAM or as distributed RAM.
RAM_1P_BRAM	A single-port RAM, implemented with a BRAM.
RAM_1P_LUTRAM	A single-port RAM, implemented as distributed RAM.
RAM_2P	A dual-port RAM, using separate read and write ports. High-Level Synthesis will determine whether to implement this in the RTL with a BRAM or as distributed RAM.

Table 3: **Storage Cores *(Cont'd)***

Core	Description
RAM_2P_BRAM	A dual-port RAM, using separate read and write ports, implemented with a BRAM.
RAM_2P_LUTRAM	A dual-port RAM, using separate read and write ports, implemented as distributed RAM.
RAM_T2P_BRAM	A true dual-port RAM, with support for both read and write on both the input and output side, implemented with a BRAM.
RAM_2P_1S	A dual-port asynchronous RAM: implemented in LUTs.
ROM_1P	A single-port ROM. High-Level Synthesis will determine whether to implement this in the RTL with a BRAM or with LUTs.
ROM_1P_BRAM	A single-port ROM, implemented with a BRAM.
ROM_1P_LUTRAM	A single-port ROM, implemented as distributed ROM.
ROM_1P_1S	A single-port asynchronous ROM: implemented in LUTs.
ROM_2P	A dual-port ROM. High-Level Synthesis will determine whether to implement this in the RTL with a BRAM or as distributed ROM.
ROM_2P_BRAM	A dual-port ROM implemented with a BRAM.
RAM_2P_LUTRAM	A dual-port ROM implemented as distributed ROM.

Connector Cores

Table 4 lists the core used to implement connectivity within the design. This includes direct connections and streaming storage elements.

Table 4: **Connector Cores**

Core	Description
Mux	Multiplexor.

Adapter Cores

Table 5 lists the cores that implement interfaces used to connect the top-level design when IP is generated. These interfaces are implemented in the RTL wrapper used in the IP generation flow in EDK.

Table 5: **Adapter Cores**

Core	Description
FSL	Standard Xilinx FSL interface.
NPI64M	Native multi-port memory controller interface.
PLB46S	Standard PLB46 slave interface.
PLB46M	Standard PLB46 master interface.

Table 5: **Adapter Cores *(Cont'd)***

Core	Description
AXI4LiteS	AXI4 Lite slave interface.
AXI4M	AXI4 master interface.
AXI4Stream	AXI4 stream interface.

Floating Point Cores

High-Level Synthesis supports the following floating point cores for each Xilinx device. If no floating point core exists for an operator or function, High-Level Synthesis will not be able to synthesis the floating point operator and synthesis will halt.

Table 6: **Floating Point Cores**

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
FRSqrt	X	X	X	-	-	-
FRSqrt_nodsp	X	X	X	-	-	-
FRSqrt_fulldsp	X	X	X	-	-	-
FRecip	X	X	X	-	-	-
FRecip_nodsp	X	X	X	-	-	-
FRecip_fulldsp	X	X	X	-	-	-
FSqrt	X	X	X	X	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X

Table 6: **Floating Point Cores *(Cont'd)***

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X
DRSqrt	X	X	X	X	X	X
DRecip	X	X	X	-	-	-
DSqrt	X	X	X	-	-	-

High-Level Synthesis Coding Style Guide

Preface

The preface includes the syntax conventions used in this document.

Conventions

The following conventions are used in document:

Table 4-1: **Syntax Conventions**

Convention	Description
`Command`	A command syntax, menu element, or a keyboard key.
`<variable>`	A user-defined value.
`choice1 | choice2`	A choice of alternatives. The underlined choice is the default.
`[option]`	An optional object.
`{repeat}`	An object repeated 0 or more times.
Ctrl+c	A keyboard combination, such as holding down the Ctrl key and pressing c.
Menu>Item	A path to a menu command, such as Item cascading from Menu.
RMB	Right Mouse Button. Gives access to context-sensitive menu in the GUI.

`<variable> ::= choice`
`$ bash_command`
`% tcl_command`
	Syntax and scripting examples (bash shell, Perl script, Tcl script).

Introduction

This coding style guide explains how you can write C code (including C++ and SystemC) for implementation on a Xilinx® FPGA device. The first step is to synthesize the C code to a register transfer level (RTL) description using Vivado™ High-Level Synthesis (HLS). The RTL design is then synthesized into Xilinx gate-level primitives.

For more details on Vivado HLS and the complete tool flow to implementation on a Xilinx FPGA, see Chapter 2, High-Level Synthesis User Guide chapter.

The initial chapters of this document explain the basics of C programming with Vivado HLS and how the HLS tool synthesizes various constructs in the C programming language into a hardware implementation. The following chapter presents guidelines for extensions to the C language: C++ and SystemC (a class library of C++ routines used for modeling hardware behavior and available from www.accellera.org).

Note: Because statements about the C language also apply to C++ and SystemC, the term C code is used throughout this document to imply code written in C, C++ or SystemC, unless specifically noted.

Algorithms written in C code are widely used in many applications and execute on many different targets, including standard microprocessors (CPUs), graphics processors (GPUs), microcontrollers used in real-time-operating-systems (RTOS) and digital signal processors (DSPs). In all cases the compiled C code executes with adequate performance. For high performance operation the C code is optimized for the target device. This document explains how modifying the code can improve the quality and performance of the hardware.

Coding Examples

Each of the numbered coding examples in this document are provided as part of the Vivado HLS release. The coding examples can be accessed in the following manner:

- From the **Browse Examples** option in the Vivado HLS start-up page.
- In the `examples/coding` directory available in the Vivado HLS installation area.

Each example directory has the same name as the top-level function for synthesis.

You can open the coding examples in the Vivado HLS GUI or use the provided Tcl script on the command prompt.

The examples in this coding guide often refer to an associated header file. Some examples show the header file: you can view all the header files in the example directory.

Note: Header files typically define the data types for the top-level function and test bench.

C for Synthesis

The top-level of every C program is the `main()` function. In HLS any function below the level of `main()` can be synthesized. In Vivado HLS, the function to be synthesized is referred to as the *top-level function,* or *design file,* and any functions above this are collectively referred to as the *test bench*. The test bench is used to validate the behavior of the top-level function to be synthesized.

The Top-Level Design

Generally, it is good design practice to separate the top-level function for synthesis from the test bench and to make use of header files.

- The test bench typically contains operations that cannot be synthesized into hardware, such as file I/O accesses to the disk.
- Header files allow definitions used in the test bench and design files to be shared and updated.
- Example 2-1 shows a design where function `hier_func` calls two sub-functions:
  - `sumsub_func` to perform addition and subtraction.
  - `shift_func` to perform shift.
- The types `din_t`, `dint_t` and `dout_t` are defined in the header file `hier_func.h`, which is also described.

```
#include "hier_func.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
     *outSum = *in1 + *in2;
     *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
     *outA = *in1 >> 1;
     *outB = *in2 >> 2;
}

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D)
{
     dint_t apb, amb;

     sumsub_func(&A,&B,&apb,&amb);
     shift_func(&apb,&amb,C,D);
}
```

Example 4-1: **Hierarchical Design Example**

The top-level function can contain multiple sub-functions; however, there can only be single top-level function for synthesis. To synthesize multiple functions group them into a single top-level function.

To synthesize function `hier_func`, the file shown in Example 2-1 can be added to an Vivado HLS project as a design file and the top-level function specified as `hier_func`. As described in later sections, the arguments to the top-level function (`A`, `B`, `C` and `D` in Example 2-1) are synthesized into RTL ports and the functions within the top-level (`sumsub_func` and `shift_func` in Example 2-1) are synthesized into hierarchical blocks.

The header file for Example 2-1, `hier_func.h`, this example shows how to use macros and the use of `typedef` statements can make the code more portable and readable. Later sections show how the `typedef` statement allows the types and hence the bit-width of the data path to be refined for both area and performance improvements in the final FPGA implementation.

```
#ifndef _HIER_FUNC_H_
#define _HIER_FUNC_H_

#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D);

#endif
```

Example 4-2: **Hierarchical Design Example Header File**

The header file includes some definitions, such as `NUM_TRANS`, which are not required in the design file. These are used by the test bench which includes the same header file.

The Test Bench

The first step in the synthesis of any block is to validate that the C function is correct. This is performed by the test bench and writing a good test bench can greatly increase designer productivity.

C functions execute in orders of magnitude faster than RTL simulations. Using C to develop and validate the algorithm prior to synthesis is more productive than developing at RTL.

- The key to taking advantage of C development times is to have a test bench that checks the results of the function against known good results. This allows any code changes to be validated before synthesis: the algorithm is known to be correct.
- Vivado HLS can re-use the C test bench to verify the RTL design (no RTL test bench needs to be created when using Vivado HLS). If the test bench checks the results from the top-level function, the RTL can be automatically verified by simulation.Example 2-3 shows the test bench for the design that was shown in Example 2-1.

```
#include "hier_func.h"

int main() {
      // Data storage
      int a[NUM_TRANS], b[NUM_TRANS];
      int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
      int c[NUM_TRANS], d[NUM_TRANS];

      //Function data (to/from function)
      int a_actual, b_actual;
      int c_actual, d_actual;

      // Misc
      int    retval=0, i, i_trans, tmp;
      FILE *fp;

      // Load input data from files
      fp=fopen("tb_data/inA.dat","r");
      for (i=0; i<NUM_TRANS; i++){
            fscanf(fp, "%d", &tmp);
            a[i] = tmp;
      }
      fclose(fp);

      fp=fopen("tb_data/inB.dat","r");
      for (i=0; i<NUM_TRANS; i++){
            fscanf(fp, "%d", &tmp);
            b[i] = tmp;
      }
      fclose(fp);

      // Execute the function multiple times (multiple transactions)
      for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

            //Apply next data values
            a_actual = a[i_trans];
            b_actual = b[i_trans];

            hier_func(a_actual, b_actual, &c_actual, &d_actual);

            //Store outputs
            c[i_trans] = c_actual;
            d[i_trans] = d_actual;
      }

      // Load expected output data from files
      fp=fopen("tb_data/outC.golden.dat","r");
      for (i=0; i<NUM_TRANS; i++){
```

```
        fscanf(fp, "%d", &tmp);
        c_expected[i] = tmp;
    }
    fclose(fp);

    fp=fopen("tb_data/outD.golden.dat","r");
    for (i=0; i<NUM_TRANS; i++){
        fscanf(fp, "%d", &tmp);
        d_expected[i] = tmp;
    }
    fclose(fp);

    // Check outputs against expected
    for (i = 0; i < NUM_TRANS-1; ++i) {
        if(c[i] != c_expected[i]){
            retval = 1;
        }
        if(d[i] != d_expected[i]){
            retval = 1;
        }
    }

    // Print Results
    if(retval == 0){
        printf("    *** *** *** *** \n");
        printf("    Results are good \n");
        printf("    *** *** *** *** \n");
    } else {
        printf("    *** *** *** *** \n");
        printf("    Mismatch: retval=%d \n", retval);
        printf("    *** *** *** *** \n");
    }

    // Return 0 if outputs are correct
    return retval;
}
```

Example 4-3: **Test Bench Example**

Creating of Productive Test Bench

Example 2-3 highlights some of the attributes of a productive test bench; such as:

- The top-level function for synthesis (`hier_func`) is executed for multiple transactions, as defined by macro `NUM_TRANS` (specified in the header file, Example 2-2), allowing for many different data values to be applied and verified. The test bench is only as good as the variety of tests it performs.
- The function outputs are compared against known good values. The known good values are read from a file in this example, but could also be computed as part of the test bench.
- The return value of `main()` function is set to zero if the results are correctly verified and to a non-zero value if the results *do not* match known good values.

TIP: *If the test bench does not return a value of 0, the RTL verification performed by Vivado HLS reports a simulation failure. To take full advantage of the automatic RTL verification, check the results in the test bench and return a 0 if the test bench has verified the results are correct.*

A test bench which exhibits these attributes quickly tests and validates any changes made to the C functions prior to synthesis and is re-usable at RTL, allowing easier verification of the RTL.

Design Files and Test Bench Files

Because Vivado HLS re-uses the C test bench for RTL verification it requires that the test bench and any associated files denoted as test bench files when they are added to the Vivado HLS project.

Files associated with the test bench are any files accessed by the test bench and required for the test bench to operate correctly. Examples of such files are the data files `inA.dat`, `inB.dat`, and so forth in Example 2-3: you must also add these to the Vivado HLS project as test bench files.

The requirement for identifying test bench files in an Vivado HLS project does not impose a requirement that the design and test bench to be in separate files (although it is recommended).

The same design from Example 2-1 is repeated below in Example 2-4. The only difference is that the top-level function is renamed `hier_func2`, to differentiate the examples.

Using the same header file and test bench (other than the change from `hier_func` to `hier_func2`), the only changes required in Vivado HLS to synthesize function `sumsum_func` as the top-level function are:

- Set `sumsub_func` as the top-level function in the Vivado HLS project.
- Add the file in Example 2-4 as both a design file *and* project file: the level above `sumsub_func`, function `hier_func2`, is now part of the test bench and must be included in the RTL simulation.

Even though function `sumsub_func` is not explicitly instantiated inside the `main()` function, the remainder of the functions (`hier_func2` and `shift_func`) are confirming it is operating correctly and thus are part of the test bench.

```
#include "hier_func2.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
     *outSum = *in1 + *in2;
     *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
```

```
{
     *outA = *in1 >> 1;
     *outB = *in2 >> 2;
}

void hier_func2(din_t A, din_t B, dout_t *C, dout_t *D)
{
     dint_t apb, amb;

     sumsub_func(&A,&B,&apb,&amb);
     shift_func(&apb,&amb,C,D);
}
```

Example 4-4: **New Top-Level**

Combining Test Bench and Design Files

It is also possible to include the design and test bench into a single design file. Example 2-5 has the same functionality as the Example 2-1 through Example 2-3, except everything is captured in a single file (function `hier_func` is renamed `hier_func3` to ensure the examples are unique).

IMPORTANT: If the test bench and design are in a single file, you must add the file to a Vivado HLS project as both a design file and a test bench file.

```
#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
     *outSum = *in1 + *in2;
     *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
     *outA = *in1 >> 1;
     *outB = *in2 >> 2;
}

void hier_func3(din_t A, din_t B, dout_t *C, dout_t *D)
{
     dint_t apb, amb;

     sumsub_func(&A,&B,&apb,&amb);
     shift_func(&apb,&amb,C,D);
}

int main() {
     // Data storage
```

```
    int a[NUM_TRANS], b[NUM_TRANS];
    int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
    int c[NUM_TRANS], d[NUM_TRANS];

    //Function data (to/from function)
    int a_actual, b_actual;
    int c_actual, d_actual;

    // Misc
    int  retval=0, i, i_trans, tmp;
    FILE *fp;
    // Load input data from files
    fp=fopen("tb_data/inA.dat","r");
    for (i=0; i<NUM_TRANS; i++){
          fscanf(fp, "%d", &tmp);
          a[i] = tmp;
    }
    fclose(fp);

    fp=fopen("tb_data/inB.dat","r");
    for (i=0; i<NUM_TRANS; i++){
          fscanf(fp, "%d", &tmp);
          b[i] = tmp;
    }
  fclose(fp);

// Execute the function multiple times (multiple transactions)
for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

          //Apply next data values
          a_actual = a[i_trans];
          b_actual = b[i_trans];

          hier_func3(a_actual, b_actual, &c_actual, &d_actual);

          //Store outputs
          c[i_trans] = c_actual;
          d[i_trans] = d_actual;
    }

    // Load expected output data from files
    fp=fopen("tb_data/outC.golden.dat","r");
    for (i=0; i<NUM_TRANS; i++){
          fscanf(fp, "%d", &tmp);
          c_expected[i] = tmp;
    }
    fclose(fp);

    fp=fopen("tb_data/outD.golden.dat","r");
    for (i=0; i<NUM_TRANS; i++){
          fscanf(fp, "%d", &tmp);
          d_expected[i] = tmp;
    }
    fclose(fp);

    // Check outputs against expected
    for (i = 0; i < NUM_TRANS-1; ++i) {
          if(c[i] != c_expected[i]){
                retval = 1;
```

```
        }
        if(d[i] != d_expected[i]){
            retval = 1;
        }
    }

    // Print Results
    if(retval == 0){
        printf("    *** *** *** *** \n");
        printf("    Results are good \n");
        printf("    *** *** *** *** \n");
    } else {
        printf("    *** *** *** *** \n");
        printf("    Mismatch: retval=%d \n", retval);
        printf("    *** *** *** *** \n");
    }

    // Return 0 if outputs are correct
    return retval;
}
```

Example 4-5: **Test Bench and Top-Level Design**

Top-Level Arguments: RTL Interface Ports

When the top-level function is synthesized the arguments (or parameters) to the function are synthesized into RTL ports. This process is called interface synthesis.

Interface Synthesis

The code shown in Figure 2-6 can be used to provide a comprehensive overview interface synthesis. In this example, there are two pass-by-value inputs, (`in1` and `in2`), a pointer (`sum`) which is both read from and written to, and a function return (the value of `temp`).

```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

    dout_t temp;

    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return  temp;
}
```

Example 4-6: **Interface Synthesis Example**

By default, the design is synthesized into an RTL block with the ports shown in Figure 2-1.

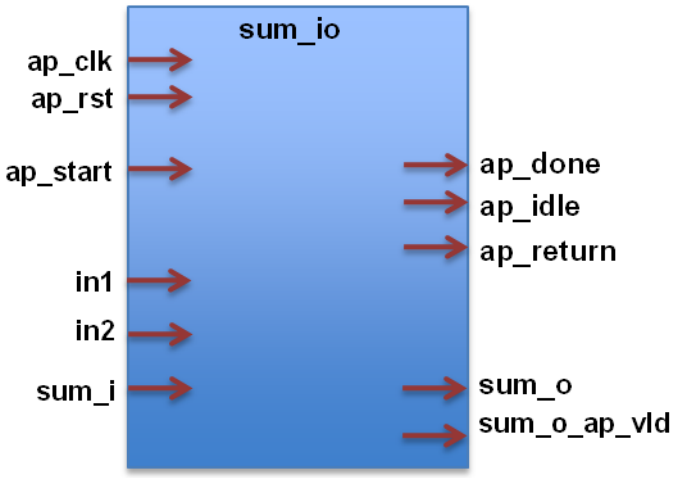

Figure 4-1: **RTL Ports After Default Interface Synthesis**

Vivado HLS has the following actions on ports. An explanation of the ports is as follows:

- Adds clock and reset port to the design.
- Adds design-level handshake signals by default: ports `ap_start`, `ap_done` and `ap_idle`.
- If the function has a return value, adds output port `ap_return` to the RTL interface.
- Vivado HLS both reads from and writes to function arguments, then synthesizes them into separate input and output ports (`sum_i` and `sum_o` in Figure 2-1).
- Vivado HLS by default, synthesizes input pass-by-value arguments and pointers as simple wire ports with no associated handshaking signal.
- By default, output pointers synthesizes with an associated output valid signal to indicate when the output data is valid.

When Vivado HLS synthesizes the RTL ports it automatically creates the necessary hardware to read and write to the ports whether it takes a single cycle or multiple cycles. For the code shown in Example 2-6 the timing behavior is shown in Figure 2-2 (assuming the target technology and clock frequency allow a single addition per clock cycle).

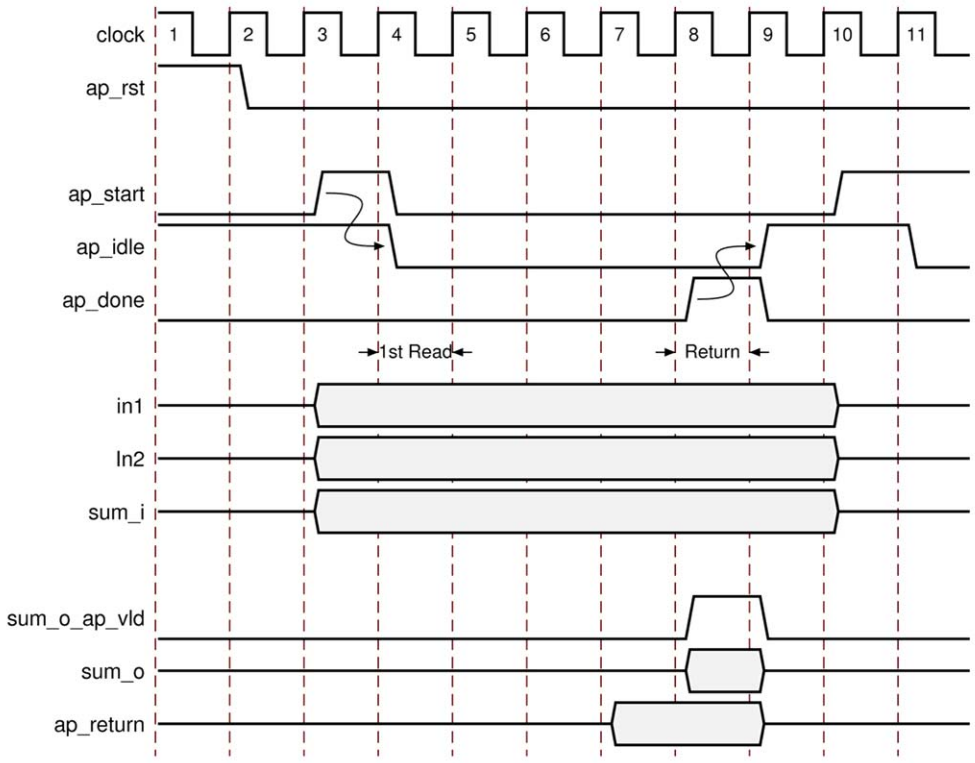

Figure 4-2: **RTL Port Timing with Default Synthesis**

- The design starts when `ap_start` is asserted high.
- The `ap_idle` signal is asserted low to indicate the design is operating.
- The input data is read at any clock after the first cycle (Vivado HLS will automatically schedule when the reads occur).
- When output `sum` is calculated, the associated output handshake (`sum_o_ap_vld`) indicates the data is valid.
- When the function completes, `ap_done` is asserted: this also indicates the data on `ap_return` is valid.
- Port `ap_idle` is asserted high to indicate the design is waiting start again.

Chapter 2, High-Level Synthesis User Guide provides a complete explanation of interface synthesis and the various options. The important points to understand here are:

- Interface synthesis automatically handles the data sequencing to and from the design: you just need to select the appropriate interface.
- Many types of interfaces can be synthesized: wire ports, single and two-way handshakes, RAM access ports and FIFO ports among others.

- Many different types of interface can be synthesized from the same source code. If the same code is synthesized with `in1`, `in2`, and `sum` specified as two-way handshakes, the RTL ports would be as shown in Figure 2-3.

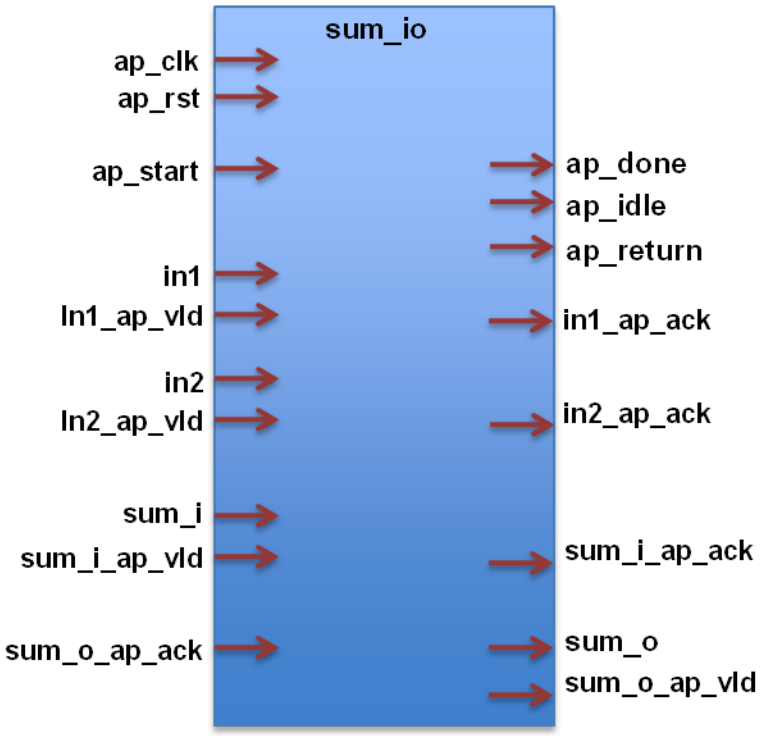

Figure 4-3: **RTL Ports After Specified Interface Synthesis**

The rest of this section describes issues related to the how the coding style can influence the implementation of RTL ports.

Pointers

Pointers can be used as arguments to the top-level function. It is important to understand how pointers are implemented during synthesis as they can sometimes introduce issues in achieving the desired RTL interface and design after synthesis.

Basic Pointers

A function with basic pointers on the top-level interface, such as shown in Example 2-7, produces no issues for HLS. The pointer can be synthesized to either a simple wire interface or an interface protocol using handshakes.

TIP: To be synthesized as a FIFO interface, a pointer must be read-only or write-only.

```
#include "pointer_basic.h"

void pointer_basic (dio_t *d) {
     static dio_t acc = 0;
```

```
    acc += *d;
    *d  = acc;
}
```

Example 4-7: **Basic Pointer Interface**

When used with the test bench shown here in Example 2-8.

```
#include "pointer_basic.h"

int main () {
    dio_t d;
    int i, retval=0;
    FILE        *fp;

    // Save the results to a file
    fp=fopen("result.dat","w");
    printf(" Din Dout\n", i, d);

    // Create input data
    // Call the function to operate on the data
    for (i=0;i<4;i++) {
        d = i;
        pointer_basic(&d);
        fprintf(fp, "%d \n", d);
        printf("  %d   %d\n", i, d);
    }
    fclose(fp);

    // Compare the results file with the golden results
    retval = system("diff --brief -w result.dat result.golden.dat");
    if (retval != 0) {
        printf("Test failed!!!\n");
        retval=1;
    } else {
        printf("Test passed!\n");
    }

    // Return 0 if the test
    return retval;
}
```

Example 4-8: **Basic Pointer Interface Test Bench**

The C function and RTL simulation will verify the correct operation (although not all possible cases) with this simple data set:

```
Din Dout
  0   0
  1   1
  2   3
  3   6
Test passed!
```

Pointer Arithmetic

When pointer arithmetic is introduced it limits the possible interfaces that can be synthesized in RTL. Example 2-9 shows the same code but this time some simple pointer arithmetic is used to accumulate the data values (starting from the 2nd value).

```
#include "pointer_arith.h"

void pointer_arith (dio_t *d) {
     static int acc = 0;
     int i;

     for (i=0;i<4;i++) {
            acc += *(d+i+1);
            *(d+i) = acc;
     }
}
```

Example 4-9: **Interface with Pointer Arithmetic**

Example 2-10 shows the test bench that supports this example. Because the loop to perform the accumulations is now inside function `pointer_arith`, the test bench populates the address space, specified by array `d[5]`, with the appropriate values.

```
#include "pointer_arith.h"

int main () {
     dio_t d[5], ref[5];
     int i, retval=0;
     FILE        *fp;

     // Create input data
     for (i=0;i<5;i++) {
            d[i] = i;
            ref[i] = i;
     }

     // Call the function to operate on the data
     pointer_arith(d);

     // Save the results to a file
     fp=fopen("result.dat","w");
     printf(" Din Dout\n", i, d);
     for (i=0;i<4;i++) {
            fprintf(fp, "%d \n", d[i]);
            printf("  %d   %d\n", ref[i], d[i]);
     }
     fclose(fp);

     // Compare the results file with the golden results
     retval = system("diff --brief -w result.dat result.golden.dat");
     if (retval != 0) {
            printf("Test failed!!!\n");
            retval=1;
     } else {
            printf("Test passed!\n");
```

```
    }

    // Return 0 if the test
    return retval;
}
```

Example 4-10: **Test Bench for Pointer Arithmetic Function**

When simulated, this results in the following output:

```
Din Dout
  0   1
  1   3
  2   6
  3   10
Test passed!
```

The problem with the pointer arithmetic is that it does not access the pointer data in sequence. Wire, handshake or FIFO interfaces have no way of accessing data out of order:

- A wire interface reads data when the design is ready to consume the data or write the data when the data is ready.
- Handshake and FIFO interfaces read and write when the control signals permit the operation to proceed.

In both cases, the data must arrive (and is written) in order, starting from element zero. In Example 2-9 the code states the first data value read is from index 1 (`i` starts at 0, 0+1=1): this is the 2nd element from array `d[5]` in the test bench.

When this is implemented in hardware, this requires some form of data indexing. This is not supported with wire, handshake or FIFO interfaces. The code in Example 2-9 can only be synthesized with an `ap_bus` interface: this interface supplies an address with which to index the data when the data is accessed (read or write).

Alternatively the code must be modified as shown in Example 2-11, with an array on the interface instead of a pointer. This can be implemented in synthesis with a RAM (`ap_memory`) interface, which has the capability of indexing the data with an address and can perform out-of-order, or non-sequential accesses.

Wire, handshake or FIFO interfaces can only be used on streaming data and therefore cannot be used in conjunction with pointer arithmetic (unless it indexes the data starting at zero and then proceeds sequentially).

More details on the `ap_bus` and `ap_memory` interface types are available in Chapter 2, High-Level Synthesis User Guide and Chapter 1, High-Level Synthesis Command Reference Guide.

```
#include "array_arith.h"

void array_arith (dio_t d[5]) {
    static int acc = 0;
```

```
        int i;

        for (i=0;i<4;i++) {
               acc += d[i+1];
               d[i] = acc;
        }
}
```

Example 4-11: **Array Arithmetic**

Multi-Access Pointer Interfaces: Streaming Data

Designs which use pointers in the argument list of the top-level function need special consideration when multiple accesses are performed using pointers. Multiple accesses occur when a pointer is read from or written to, multiple times in the same function.

The issues which arise are that:

- It is a requirement to use the volatile qualifier on any function argument accessed multiple times.
- On the top-level function, any such argument must have the number of accesses on the port interface specified if verifying the RTL using co-simulation within Vivado HLS.
- Be sure to validate the C prior to synthesis to confirm the intent and the C model is correct.

RECOMMENDED: If modeling the design requires that an function argument be accessed multiple times it is recommended to model the design using streams, as explained in section Designing with Streaming Data [ADD REFERENCE]. Using streams ensures none of the issues detailed in this section will be encountered.

The section shows, using example design pointer_stream_bad, why the volatile qualifier is required when accessing pointers multiple times within the same function and also highlights, using example design pointer_stream_better, why any design which have such pointers on the top-level interface should be verified with a C test bench to ensure the intended behavior is correctly modelled.

In Example 2-12, input pointer `d_i` is read from four times and output `d_o` is written to twice, with the intent that the accesses will be implemented by FIFO interfaces (streaming data into and out of the final RTL implementation).

```
#include "pointer_stream_bad.h"

void pointer_stream_bad ( dout_t *d_o,  din_t *d_i) {
     din_t acc = 0;

     acc += *d_i;
     acc += *d_i;
     *d_o = acc;
```

```
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```

Example 4-12: **Multi-Access Pointer Interface**

The test bench to verify this design is shown in Example 2-13.

```
#include "pointer_stream_bad.h"

int main () {
    din_t d_i;
    dout_t d_o;
    int retval=0;
    FILE *fp;

    // Open a file for the output results
    fp=fopen("result.dat","w");

    // Call the function to operate on the data
    for (d_i=0;d_i<4;d_i++) {
        pointer_stream_bad(&d_o,&d_i);
        fprintf(fp, "%d %d\n", d_i, d_o);
    }
    fclose(fp);

    // Compare the results file with the golden results
    retval = system("diff --brief -w result.dat result.golden.dat");
    if (retval != 0) {
        printf("Test failed  !!!\n");
        retval=1;
    } else {
        printf("Test passed !\n");
    }

    // Return 0 if the test
    return retval;
}
```

Example 4-13: **Multi-Access Pointer Test Bench**

Understanding Volatile Data

The code in Example 2-12 is written with *intent* that input pointer `d_i` and output pointer `d_o` will be implemented in RTL as FIFO (or handshake) interfaces which will ensure:

- Upstream producer blocks will supply new data each time a read is performed on RTL port `d_i`.
- Downstream consumer blocks will accept new data each time there is a write to RTL port `d_o`.

However, when this code is compiled by standard C compilers, the multiple accesses to each pointer will be reduced to a single access: as far as the compiler is concerned, there is no indication that the data on d_i changes during the execution of the function and only the final write to d_o is relevant (the other writes will be over-written by the time the function completes).

Vivado HLS matches the behavior of the gcc compiler and optimizes these reads and writes into a single read operation and a single write operation. When the RTL is examined, there will only be a single read and write operation on each port.

The fundamental issue with this design is that the test bench and design do not adequately model how the designer expects the RTL ports to be implemented:

- The designer expects RTL ports which read and write multiple times during a transaction (and can stream the data in and out).
- The test bench only supplies a single input value and only returns a single output value. A C simulation of Example 2-12 would show the following results, which demonstrates each input is being accumulated 4 times, but it's the same value being read once and accumulated each time: not 4 separate reads.

```
Din Dout
0   0
1   4
2   8
3   12
```

This design can be made read and write to the RTL ports multiple times by using the volatile qualifier as shown below in Example 2-14.

The volatile qualifier tells the C compiler, and Vivado HLS, to make no assumptions about the pointer accesses: the data is volatile, may change and pointer accesses should not be optimized.

```
#include "pointer_stream_better.h"

void pointer_stream_better ( volatile dout_t *d_o,  volatile din_t *d_i) {
     din_t acc = 0;

     acc += *d_i;
     acc += *d_i;
     *d_o = acc;
     acc += *d_i;
     acc += *d_i;
     *d_o = acc;
}
```

Example 4-14: **Multi-Access Volatile Pointer Interface**

Example 2-14 will simulate the same as Example 2-12 but the volatile qualifier will prevent pointer access optimizations and result in an RTL design which will perform the expected four reads on input port d_i and two writes to output port d_o.

However, even if the volatile keyword is used, this coding style (accessing a pointer multiple times) still has an issue in that the function and test bench do not adequately model multiple distinct reads and writes.

In this case, four reads will be performed, but again, the same data will be read four times. There will be two separate writes, each with the correct data but the test bench will only capture data for the final write. (The intermediate accesses can be seen by enabling `cosim_design` to create a trace file during RTL simulation and viewing the VCD file).

Note: Example 2-14 can be implemented with wire interfaces, but if a FIFO interface is specified Vivado HLS will create an RTL test bench to stream new data on each read: since there is no new data available from the test bench, the RTL will fail to verify. The issue here is that the test bench does not correctly model the reads and writes correctly.

Modeling Streaming Data Interfaces

Unlike software, the concurrent nature of hardware systems allows them to take advantage of streaming data, where data is continuously supplied to the design and the design continuously outputs data: an RTL design can accept new data before the design has finished processing the existing data.

As the Example 2-14 has shown, modeling streaming data in software is non-trivial, especially when writing software to model an existing hardware implementation (where the concurrent/streaming nature already exists and needs to be modeled).

There are a number of approaches which can be taken here:

- Simply add the volatile qualifier as shown in Example 2-14. The test bench will not model unique reads and writes and RTL simulation using the original C test bench may fail, but viewing the VCD waveforms will show the correct reads and writes are being performed.
- Modify the code to model explicit unique reads and writes. This is shown next in Example 2-15.
- Modify the code to using a streaming data type. A streaming data type allows hardware using streaming data to be accurately modeled. This is discussed in Chapter 2, High-Level Synthesis User Guide.

The code shown in Example 2-15 has been updated to ensure it will read four unique values from the test bench and write two unique values. Since the pointer accesses are sequential and start at location zero, a streaming interface type can be used during synthesis.

```
#include "pointer_stream_good.h"

void pointer_stream_good ( volatile dout_t *d_o,  volatile din_t *d_i) {
     din_t acc = 0;

     acc += *d_i;
     acc += *(d_i+1);
     *d_o = acc;
```

```
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;
}
```

Example 4-15: **Explicit Multi-Access Volatile Pointer Interface**

The test bench is updated to model the fact that the function will read four unique values in each transaction. This new test bench only models a single transaction: to model multiple transactions, the input data set would need to be increased to and the function called multiple times.

```
#include "pointer_stream_good.h"

int main () {
    din_t d_i[4];
    dout_t d_o[4];
        int i, retval=0;
        FILE        *fp;

    // Create input data
    for (i=0;i<4;i++) {
        d_i[i] = i;
    }

    // Call the function to operate on the data
    pointer_stream_good(d_o,d_i);

    // Save the results to a file
    fp=fopen("result.dat","w");
    for (i=0;i<4;i++) {
        if (i<2)
    fprintf(fp, "%d %d\n", d_i[i], d_o[i]);
        else
    fprintf(fp, "%d \n", d_i[i]);
    }
    fclose(fp);

    // Compare the results file with the golden results
    retval = system("diff --brief -w result.dat result.golden.dat");
    if (retval != 0) {
        printf("Test failed  !!!\n");
        retval=1;
    } else {
        printf("Test passed !\n");
    }

    // Return 0 if the test
    return retval;
}
```

Example 4-16: **Explicit Multi-Access Volatile Pointer Test Bench**

The test bench will validate the algorithm with the following results, showing there are two outputs from a single transaction and they are an accumulation of the first two input reads, plus an accumulation of the next two input reads and the previous accumulation:

```
Din Dout
0   1
1   6
2
3
```

The final issue to be aware of when pointers are accessed multiple time at the function interface is RTL simulation modeling.

Multi-Access Pointers and RTL Simulation

To verify the RTL with `cosim_design`, Vivado HLS creates a SystemC wrapper with adapters around the RTL and instantiates this wrapper into the existing C test bench, as shown in Figure 2-4.

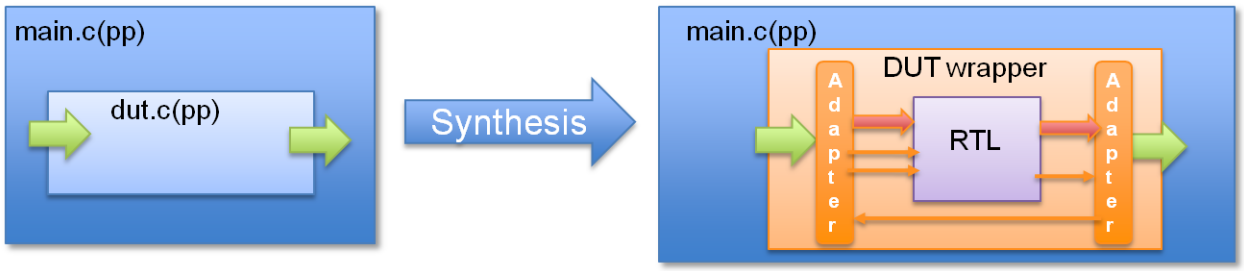

Figure 4-4: **`Cosim_design` Wrapper Overview**

The wrapper created by Vivado HLS models any required handshakes on the RTL interface and as such must ensure the input values to the DUT, presented by the test bench, are ready when required by the RTL design. This requires storage in the adapter.

When pointers on the interface are accessed multiple times, to read or write, Vivado HLS cannot determine from the function interface how many reads or writes are performed. Neither of the arguments in the function interface informs Vivado HLS how many values will be read or written.

```
void pointer_stream_good ( volatile dout_t *d_o,  volatile din_t *d_i)
```

Example 4-17: **Volatile Pointer Interface**

Unless something on the interface informs Vivado HLS as to how many values are required, such the maximum size of an array, Vivado HLS will assume a single value and only create simulation wrappers for a single input and single output.

If the RTL ports are actually reading or writing multiple values, this will result in the RTL `cosim_design` simulation stalling: the wrapper is modeling the producer and consumer blocks which will be connected to the RTL design, and if it only models a single value the RTL design will stall when trying to read or write more than one value (since there is currently no value to read or no space to write).

When multi-access pointers are used at the interface, Vivado HLS must be informed of the maximum number of reads or writes on the interface. When specifying the interface, use the depth option on the `INTERFACE` directive as shown in Figure 2-5, page 222.

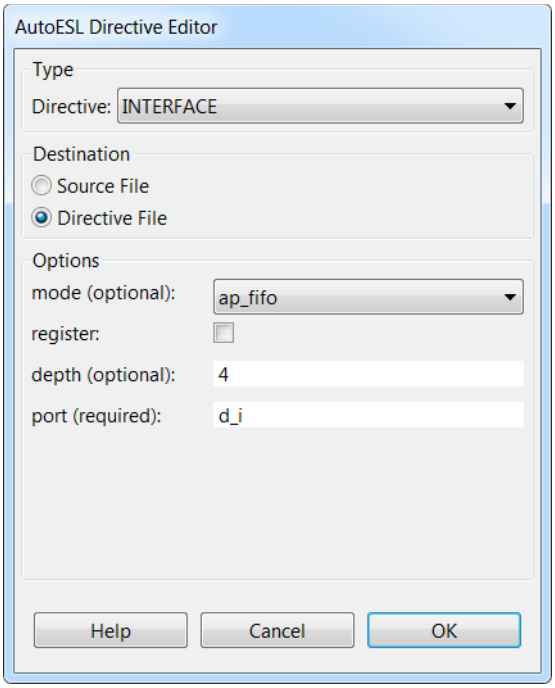

Figure 4-5: **Interface Directive Dialog: Depth Option**

In the above example, argument/port `d_i` is set to have a FIFO interface with a depth of 4, ensuring that `cosim_design` will provide enough values to correctly verify the RTL.

Arrays on the Interface

In HLS, arrays are synthesized into memory elements by default. When an array is used as an argument to the top-level function the memory is assumed to be "off-chip" and interface ports are synthesized to access the memory.

Vivado HLS has a rich feature set to configure how these ports are created.

- The memory can be specified as a single or dual port RAM.
- The interface can be specified as a FIFO interface.
- Vivado HLS array optimization directives (`partition`, `map` and `reshape`) can be used to re-configure the structure of the array and hence number of IO ports.

The primary issue which arrays on the interface will introduce into a design is creating a performance bottleneck because access to the data is limited through a memory (RAM or FIFO) port. These issues can typically be over-come with the use of directives.

The main rule for using arrays in synthesizable code, it is that arrays must be sized. If for example, the declaration `d_i[4]` in Example 2-18 is changed to `d_i[]`, Vivado HLS will issues a message that the design cannot be synthesized.

```
@E [SYNCHK-61] array_RAM.c:52: unsupported memory access on
variable 'd_i' which is (or contains) an array with unknown size at compile time.
```

Array Optimization Directives

The resource directive can be used to explicitly specify which type of RAM is used, and hence which RAM ports are created: single-port or dual-port. If no resource is specified Vivado HLS will use a single-port RAM by default and automatically use a dual-port RAM if it improves throughput or reduces latency.

The `partition`, `map` and `reshape` directives can be used to re-configure arrays on the interface. Arrays can be partitioned into multiple smaller arrays, each implemented with its own interface. This includes the ability to partition every element of the array into its own scalar element: on the function interface, this results in a unique port for every element in the array. This provides maximum parallel access but creates many more ports and may introduce routing issues in the hierarchy above.

Similarly, smaller arrays may be combined into a single larger array, resulting in a single interface. This may map better to an "off-chip" BRAM but keep in mind it may also introduce a performance bottleneck. These trade-offs can made using Vivado HLS optimization directives and do not impact coding.

RAM interfaces

The array arguments in the function shown in Example 2-18 will, by default, be synthesized into a single-port RAM interface.

```
#include "array_RAM.h"

void array_RAM (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
     int i;

     For_Loop: for (i=0;i<4;i++) {
            d_o[i] = d_i[idx[i]];
     }

}
```

Example 4-18: **RAM Interface**

A single-port RAM interface will used because the for-loop ensures only one element can be read and written in each clock cycle: there is no advantage in using a dual-port RAM interface.

If the for-loop is unrolled, Vivado HLS will automatically use a dual-port because doing so will allow multiple elements to be read at the same time and increase the throughput. The type of RAM interface can be explicitly set by applying the resource directive.

As mentioned earlier, if there are issues related to arrays on the interface, they are typically related to throughput and can be handled with optimization directives. For example, if the arrays in Example 2-18 are partitioned into individual elements and the for-loop unrolled, all four elements in each array will be accessed simultaneously.

FIFO Interfaces

Vivado HLS allows array arguments to be implemented as FIFO ports in the RTL. If a FIFO ports is to be used, be sure that the accesses to and from the array are sequential. Vivado HLS will perform analysis to confirm if the accesses are sequential.

- If Vivado HLS can verify the accesses are sequential it will implement a FIFO port.
- If Vivado HLS can determine the accesses are not sequential it will issue an error and synthesis will halt.
- If Vivado HLS cannot determine if the accesses are sequential it will issue a warning and proceed with the implementation of a FIFO port. If the accesses are in fact not sequential it will result in an RTL simulation mismatch.

Example 2-19 shows a case where Vivado HLS cannot determine if the accesses are sequential. In this example, both d_i and d_o are specified to be implemented with a FIFO interface during synthesis.

```
#include "array_FIFO.h"

void array_FIFO (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
     int i;

     // Breaks FIFO interface d_o[3] = d_i[2];
     For_Loop: for (i=0;i<4;i++) {
            d_o[i] = d_i[idx[i]];
     }
}
```

Example 4-19: **Streaming FIFO Interface**

In this case, it is the behavior of variable `idx` which determines if a FIFO interface can be successfully created or not.

- If `idx` is incremented sequentially a FIFO interface can be created.
- If random values are used for `idx`, a FIFO interface will fail when implemented in RTL.

Since there is a possibility that this interface may work, Vivado HLS will issue a message during synthesis and proceed to create a FIFO interface.

```
@W [XFORM-124] Array 'd_i': may have improper streaming access(es).
```

If the comments in Example 2-19 are removed, (“//Breaks FIFO interface”) Vivado HLS can automatically determine the accesses to the arrays are not sequential and will halt with an error message if a FIFO interface is specified.

Note: FIFO ports cannot be synthesized for arrays which are read from and written to: separate input and output arrays (as in Example 2-19) must be created.

The following general rules apply to arrays which are to be streamed (implemented with a FIFO interface):

- The array must be written/read in only one loop or function. This can be transformed into a point to point connection which matches the characteristics of FIFO links.
- The array reads must be in the same order as the array write. Random access is not supported for FIFO channels, so the array has to be used in the program following first in first out semantics.
- The index used to read and write from the FIFO has to be analyzable at compile time. Array addressing based on runtime computations cannot be analyzed for FIFO semantics and will prevent the tool from converting an array into a FIFO.

Code changes are generally not required to implement or optimize arrays in the top-level interface. The only time arrays on the interface may need coding changes is when the array is part of a struct.

Structs on the Interface

When structs are used as arguments to the top-level function, the ports created by synthesis depend on whether the struct is a pass-by-value argument or a pointer.

In this design example, `struct data_t` is defined in the header file shown in Example 2-20. This struct has two data members:

- An unsigned vector `A` of type `short` (16-bit).
- An array `B` of four `unsigned char` types (8-bit).

```
typedef struct {
     unsigned short A;
     unsigned char B[4];
     } data_t;

data_t struct_port(data_t i_val, data_t *i_pt, data_t *o_pt);
```

Example 4-20: **Struct Declaration in Header file**

In Example 2-21 the struct is used as both a pass-by-value argument (from `i_val` to the return of `o_val`) and as a pointer (`*i_pt` to `*o_pt`). Although both methods provide a similar result—passing the input to the output after an addition operation—the difference is how the pass-by-value and pointer arguments are synthesized as ports.

```
#include "struct_port.h"

data_t struct_port(
     data_t  i_val,
     data_t  *i_pt,
     data_t  *o_pt
     ) {

     data_t  o_val;
     int i;

     // Transfer pass-by-value structs
     o_val.A = i_val.A+2;
     for (i=0;i<4;i++) {
            o_val.B[i] = i_val.B[i]+2;
     }

     // Transfer pointer structs
     o_pt->A = i_pt->A+3;
     for (i=0;i<4;i++) {
            o_pt->B[i] = i_pt->B[i]+3;
     }

     return o_val;
}
```

Example 4-21: **Struct as Pass-by-Value and Pointer**

In the case of the pass-by-value input arguments, the arrays in the struct will be completely partitioned into separate elements:

- `Struct` element A will result in a 16-bit port.
- `Struct` element B will result in 4 separate 8-bit ports.

For the pointers and the function return, any arrays in the struct will be synthesized in the same manner as standard arrays and will result in memory interface:

- `Struct` element A will result in a 16-bit port.
- `Struct` element B will result in a RAM port, accessing 4 elements.

When using structs with large arrays, it may be an advantage to convert any pass-by-value structs to pointers otherwise such arrays will be completely partitioned into individual elements, each implemented with their own port. For example, if the array contains 1024 elements, it will be implemented with 1024 separate RTL ports.

There are no limitations in the size or complexity of structs which can be synthesized by Vivado HLS. There can be as many array dimensions and as many members in a struct required. The only limitation with the implementation of structs is when arrays are to be implemented as streaming (such as a FIFO interface). In this case, the same general rules which apply to arrays on the interface should be followed (FIFO Interfaces).

Types

The data types used in a C function compiled into an executable impact the accuracy of the result, the memory requirements and can impact the performance.

- A 32-bit integer int data type can hold more data and hence provide more precision than an 8-bit char type but obviously requires more storage.
- If 64-bit `long long` types are used on a 32-bit system the run time will be impacted as it will typically require multiple accesses to read and write such values.

Similarly, when the C function is to be synthesized to an RTL implementation the types impact the precision, the area and the performance of the RTL design: the data types used for variables determine the size of the operators required and hence the area and performance of the RTL.

Vivado HLS supports the synthesis of all standard C types including exact-width integer types.

- `(unsigned) char`, `(unsigned) short`, `(unsigned) int`
- `(unsigned) long`, `(unsigned) long long`
- `(unsigned) intN_t` (where `N` is 8,16,32 and 64, as defined in `stdint.h`)
- `float`, `double`

Exact-width integers types are useful for ensuring designs are portable across all types of system.

> ***Note:*** Integer type `(unsigned)long` is implemented as 64-bit on 64-bit operating systems and as 32-bit on 32-bit operating systems. Synthesis matches this behavior and will produce different sized operators, and hence different RTL designs, depending on the type of operating system on which Vivado HLS is run.
>
> Data type `(unsigned)int` or `(unsigned)int32_t` should be used instead of type `(unsigned)long` for 32-bit.
>
> Data type `(unsigned)long long` or `(unsigned)int64_t` should be used instead of type `(unsigned)long` for 64-bit.

Standard Types

Example 2-22 shows some basic arithmetic operations being performed.

```
#include "types_standard.h"

void types_standard(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
              dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
) {

     // Basic arithmetic operations
     *out1 = inA * inB;
```

```
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;

}
```

Example 4-22: **Basic Arithmetic**

The data types in Example 2-22 are defined in the header file `types_standard.h` shown in Example 2-23 and show how standard signed types, unsigned types, and with the inclusion of header file `stdint.h`, exact-width integer types can be used.

```
#include <stdio.h>
#include <stdint.h>

#define N 9

typedef char din_A;
typedef short din_B;
typedef int din_C;
typedef long long din_D;

typedef int dout_1;
typedef unsigned char dout_2;
typedef int32_t dout_3;
typedef int64_t dout_4;

void types_standard(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

Example 4-23: **Basic Arithmetic Type Definitions**

These different types result in the following operator and port sizes after synthesis:

- The multiplier used to calculate result `out1` will be a 24-bit multiplier: an 8-bit `char` type multiplied by a 16-bit `short` type requires a 24-bit multiplier. This result will be sign-extended to 32-bit to match the output port width.
- The adder used for `out2` will be 8-bit: since the output is an 8-bit `unsigned char` type, only the bottom 8-bits of `inB` (a 16-bit `short`) are added to 8-bit `char` type `inA`.
- For output `out3` (32-bit exact width type), 8-bit `char` type `inA` is sign-extended to 32-bit value and a 32-bit division operation is performed with the 32-bit (`int` type) `inC` input.
- Similarly, a 64-bit modulus operation is performed using the 64-bit `long long` type `inD` and 8-bit `char` type `inA` sign-extended to 64-bit, to create a 64-bit output result `out4`.

As the result of `out1` indicates, Vivado HLS will use the smallest operator it can and simply extend the result to match the required output bit-width. Similarly, for result `out2` even though one of the inputs is 16-bit, an 8-bit adder can be used because only an 8-bit output

is required. However, as the results for `out3` and `out4` demonstrate, if all bits are required, a full sized operator will be synthesized.

Floats and Doubles

Vivado HLS supports `float` and `double` types for synthesis. Both data types are synthesized with IEEE-754 standard compliance.

- Single-precision 32 bit: 24-bit fraction, 8-bit exponent
- Double-precision 64 bit: 53-bit fraction, 11-bit exponent

In addition to using floats and doubles for standard arithmetic operations (+, -, * etc.) floats and doubles are commonly used with the math.h (and cmath.h for C++). This section details how standard operators are supported. Refer to section HLS Math Library [ADD REFERNCE] for details on how to synthesize the C and C++ math libraries.

Example 2-24 show the header file used with Example 2-22 updated to define the data types to be `double` and `float` types.

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>

#define N 9

typedef double din_A;
typedef double din_B;
typedef double din_C;
typedef float din_D;

typedef double dout_1;
typedef double dout_2;
typedef double dout_3;
typedef float dout_4;

void types_float_double(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

Example 4-24: **Float and Double Types**

This updated header file is used with Example 2-25, where a `sqrtf()` function is used.

```
#include "types_float_double.h"

void types_float_double(
     din_A  inA,
     din_B  inB,
     din_C  inC,
     din_D  inD,
     dout_1 *out1,
     dout_2 *out2,
     dout_3 *out3,
     dout_4 *out4
     ) {
```

```
        // Basic arithmetic & math.h sqrtf()
        *out1 = inA * inB;
        *out2 = inB + inA;
        *out3 = inC / inA;
        *out4 = sqrtf(inD);

}
```

Example 4-25: **Use of Floats and Doubles**

When Example 2-25 is synthesized it will result in 64-bit double-precision multiplier, adder and divider operators: these operators will be implemented by the appropriate floating-point Xilinx CORE Generator cores.

The square-root function used `sqrtf()` will be implemented using a 32-bit single-precision floating-point core. It is worth noting that if the double-precision square-root function `sqrt()` was used, it would result in additional logic to cast to and from the 32-bit single-precision float types used for inD and `out4: sqrt()` is a double-precision (`double`) function, while `sqrtf()` is a single precision (`float`) function.

CAUTION! *In C functions, be careful when mixing float and double types as float-to-double and double-to-float conversion units will be inferred in the hardware.*

This code:

```
float foo_f    = 3.1459;
float var_f = sqrt(foo_f);
```

Would result in the following hardware:

```
wire(foo_t)
→ Float-to-Double Converter unit
→ Double-Precision Square Root unit
→ Double-to-Float Converter unit
→ wire (var_f)
```

Using a `sqrtf()` function would remove the need for the type converters in hardware, save area and improve timing.

Operations in float and double types are synthesized to a floating point operator LogicCore cores. Table 2-2 shows the cores available for each Xilinx family.

The implications from the cores shown in Table 2-2 are that if the technology does not support a particular LogicCore element, the design cannot be synthesized and Vivado HLS will halt with an error message.

Table 4-2: **Floating Point Cores**

Core	7-Series	Virtex 6	Virtex 5	Virtex 4	Spartan 6	Spartan 3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-

Table 4-2: **Floating Point Cores *(Cont'd)***

Core	7-Series	Virtex 6	Virtex 5	Virtex 4	Spartan 6	Spartan 3
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X

The cores in Table 2-2 allow the operation, in some cases, to be implemented with a core in which many DSP48s are used or none (e.g. DMul_nodsp and DMul_maxdsp). By default, Vivado HLS will implement the operation using the core with the maximum number of DSP48s. Alternatively, the Vivado HLS resource directive can be used to specify exactly which core should be used.

A final consideration to be aware of when synthesizing float and double types is that Vivado HLS will maintain the order of operations performed in the C code to ensure the results are the same as the C simulation. Due to saturation and truncation, the following are not guaranteed to be the same in single and double precision operations:

```
A=B*C;        A=B*F;
D=E*F;        D=E*C;
O1=A*D        O2=A*D;
```

With `float` and `double` types, `O1` and `O2` are not guaranteed to be the same.

TIP: *In some cases (design dependent), optimizations such as unrolling or partial unrolling of loops, may not be able to take full advantage of parallel computations as Vivado HLS will maintain the strict order of the operations when synthesizing float and double types.*

For C++ designs, Vivado HLS provides a bit-approximate implementation of the most commonly used math functions.

Composite Types

Composite data types are supported for synthesis:

- `array`
- `enum`
- `struct`
- `union`

Example 2-26 shows a header file which first defines some `enum` types, uses them in a `struct`, which is in turn used in another `struct`, allowing an intuitive description of a complex type to be captured.

In addition, Example 2-26 shows how a complex define (`MAD_NSBSAMPLES`) statement can be specified and synthesized.

```
#include <stdio.h>

enum mad_layer {
     MAD_LAYER_I   = 1,
     MAD_LAYER_II  = 2,
     MAD_LAYER_III = 3
};

enum mad_mode {
     MAD_MODE_SINGLE_CHANNEL = 0,
     MAD_MODE_DUAL_CHANNEL = 1,
     MAD_MODE_JOINT_STEREO = 2,
     MAD_MODE_STEREO = 3
};

enum mad_emphasis {
     MAD_EMPHASIS_NONE = 0,
     MAD_EMPHASIS_50_15_US = 1,
     MAD_EMPHASIS_CCITT_J_17 = 3
};

typedef   signed int mad_fixed_t;

typedef struct mad_header {
     enum mad_layer layer;
       enum mad_mode mode;
     int mode_extension;
     enum mad_emphasis emphasis;

     unsigned long long bitrate;
     unsigned int samplerate;

     unsigned short crc_check;
     unsigned short crc_target;
```

```
    int flags;
    int private_bits;

} header_t;

typedef struct mad_frame {
    header_t header;
    int options;
    mad_fixed_t sbsample[2][36][32];
} frame_t;

# define MAD_NSBSAMPLES(header)  \
  ((header)->layer == MAD_LAYER_I ? 12 :  \
    (((header)->layer == MAD_LAYER_III &&  \
      ((header)->flags & 17)) ? 18 : 36))


void types_composite(frame_t *frame);
```

Example 4-26: **Enum, Struct & Complex Define**

The `struct` and `enum` types defined in Example 2-26 are used in Example 2-27. As with standard C compilation, `enum` types are assumed to be 32-bit values and as such will result in 32-bit values after synthesis.

Example 2-27 also shows how `printf` statements will be automatically ignored during synthesis.

```
#include "types_composite.h"

void types_composite(frame_t *frame)
{
    if (frame->header.mode != MAD_MODE_SINGLE_CHANNEL) {
        unsigned int ns, s, sb;
        mad_fixed_t left, right;

        ns = MAD_NSBSAMPLES(&frame->header);
        printf("Samples from header %d \n", ns);

        for (s = 0; s < ns; ++s) {
            for (sb = 0; sb < 32; ++sb) {
                left  = frame->sbsample[0][s][sb];
                right = frame->sbsample[1][s][sb];
                frame->sbsample[0][s][sb] = (left + right) / 2;
            }
        }
        frame->header.mode = MAD_MODE_SINGLE_CHANNEL;
    }
}
```

Example 4-27: **Use Complex Types**

In Example 2-28 a union is created with a `double` and a `struct`. Unlike C compilation, synthesis will not guarantee to use the same memory (in the case of synthesis,

registers) for all fields in the `union`. Vivado HLS will perform whatever optimization provides the most optimal hardware.

Note: Pointer reinterpretation is not supported for synthesis. As such, a union cannot hold pointers to different types (or arrays of different types).

```
#include "types_union.h"

dout_t types_union(din_t N, dinfp_t F)
{
     union {
            struct {int a; int b; } intval;
            double fpval;
     } intfp;
     unsigned long long one, exp;

     // Set a floating-point value in union intfp
     intfp.fpval = F;

     // Slice out lower bits and add to shifted input
     one = intfp.intval.a;
     exp = (N & 0x7FF);

     return ((exp << 52) + one) & (0x7fffffffffffffffLL);
}
```

Example 4-28: **Unions**

Type Qualifiers

The type qualifiers can have a direct impact on the hardware created by high-level synthesis. In general, the qualifiers influence the synthesis results in a predictable manner, as detailed below, however Vivado HLS is only limited by the interpretation of the qualifier as it affects functional behavior and can perform optimizations to create a more optimal hardware design. Examples of this are shown after an overview of each qualifier.

Volatile

The `volatile` qualifier impacts how many reads or writes are performed in the RTL when pointers are accessed multiple times on function interfaces. Although the `volatile` qualifier impacts this behavior in all functions in the hierarchy the impact of the `volatile` qualifier is discussed in the section on top-level interfaces (Refer to the section Understanding Volatile Data, page 217).

Statics

Static types in a function hold their value between function calls. The equivalent behavior in a hardware design is a registered variable (a flip-flop or memory). If a variable is required to be a static type for the C function to execute correctly, it will certainly be a register in the final RTL design: the value must be maintained across invocations of the function and design.

It is *not* true however to state that *only* `static` types will result in a register after synthesis. Vivado HLS will determine which variables are required to be implemented as registers in the RTL design. For example, if a variable assignment must be held over multiple cycles, Vivado HLS will create a register to hold the value, even if the original variable in the C function was *not* a static type.

Vivado HLS obeys the initialization behavior of statics and automatically assigns the value to zero, or any explicitly initialized value, to the register during initialization. This means the `static` variable will be initialized in the RTL code and in the FPGA bitstream. It does not automatically mean the variable will be re-initialized each time the reset signal is asserted.

Note: Refer to the RTL configuration (`config_rtl` command) to determine how static initialization values are implemented with regard to the system reset.

Const

A const type specifies that the value of the variable is never updated. The variable is read but never written to and therefore must be initialized. For most const variables, this will typically mean they will be reduced to constants in the RTL design (Vivado HLS will perform constant propagation and remove any unnecessary hardware).

In the case of arrays however, the `const` variable will be implemented as a ROM in the final RTL design (in the absence of any auto-partitioning performed by Vivado HLS on small arrays). Arrays specified with the `const` qualifier will, like statics, be initialized in the RTL and in the FPGA bitstream. (There is no need to reset them since they are never written to).

Vivado HLS Optimizations

Example 2-29 shows a case where Vivado HLS will implement a ROM even though the array is not specified with a `static` or `const` qualifier. This highlights how Vivado HLS will analyze the design and automatically determine the most optimal implementation: the qualifiers, or lack of them, influence but do not dictate the final RTL.

```
#include "array_ROM.h"

dout_t array_ROM(din1_t inval, din2_t idx)
{
     din1_t lookup_table[256];
     dint_t i;

     for (i = 0; i < 256; i++) {
     lookup_table[i] = 256 * (i - 128);
     }

     return (dout_t)inval * (dout_t)lookup_table[idx];
}
```

Example 4-29: **Non-static, Non-const, ROM implementation**

In the case of Example 2-29, Vivado HLS is able to determine the implementation is best served by having the variable `lookup_table` as a memory element in the final RTL. More details on how this achieved for arrays is discussed in the section Implementing ROMs.

Global Variables

Global variables can be freely used in the code and are fully synthesizable. By default however, global variables are not exposed as ports on the RTL interface. Example 2-30 helps explain how global variables are synthesized.

In Example 2-30, three global variables are used. Although this example uses arrays, all types of global variables are supported.)

- Values are read from array `Ain`.
- Array Aint is used to transform and pass values from `Ain` to `Aout`.
- The outputs are written to array `Aout`.

```
din_t Ain[N];
din_t Aint[N];
dout_t Aout[N/2];

void types_global(din1_t idx) {
     din_t i,lidx;

     // Move elements in the input array
     for (i=0; i<N; ++i) {
            lidx=i;
            if(lidx+idx>N-1)
                  lidx=N-1;
            Aint[lidx] = Ain[lidx+idx] + Ain[lidx];
     }

     // Sum to half the elements
     for (i=0; i<(N/2); i++) {
            Aout[i] = (Aint[i] + Aint[i+1])/2;
     }

}
```

Example 4-30: **Global variables**

By default, after synthesis, the only port on the RTL design will be port `idx`: global variables are not exposed as RTL ports by default. In the default case, array Ain is an internal RAM which is read from and Aout an internal RAM which is written to.

The `expose_global` option in the Vivado HLS interface configuration can be used to instruct Vivado HLS to expose global variables as ports on the RTL interface. In this case, ports will be created to access both `Ain` and `Aout` as external RAMs. In addition however, ports will also be created showing the accesses to internal RAM.

Note: When global variables are exposed, all global variables in the design, including those which only have accesses internal to the design, are exposed as RTL ports.

In summary, global variables are supported for synthesis, however a coding style which uses global variables extensively is not recommended.

Pointers

Pointers are used extensively in C code and are well supported for synthesis. The only cases where care needs to be taken when using pointers are:

- When pointers are accessed (read or written) multiple times in the same function. Refer to Multi-Access Pointer Interfaces: Streaming Data for issues related to this.
- When using arrays of pointers, each pointer must point to a scalar or a scalar array: not another pointer.
- Pointer casting is only supported when casting between standard C types, as shown.

Many previous examples have shown how C pointers can be synthesized using Vivado HLS. Synthesis support for pointers includes, as shown in Example 2-31, cases where pointers point to multiple objects.

```
#include "pointer_multi.h"

dout_t pointer_multi (sel_t sel, din_t pos) {
     static const dout_t a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
     static const dout_t b[8] = {8, 7, 6, 5, 4, 3, 2, 1};

     dout_t* ptr;
     if (sel)
            ptr = a;
     else
     ptr = b;

     return ptr[pos];
}
```

Example 4-31: **Multiple Pointer Targets**

Double-pointers are also supported for synthesis (Example 2-32). If a double-pointer is used in multiple functions, Vivado HLS will inline all functions in which it is used. If multiple functions are inlined, it may cause an increase in run time.

```
#include "pointer_double.h"

data_t sub(data_t ptr[10], data_t size, data_t**flagPtr)
{
     data_t x, i;

     x = 0;
     // Sum x if AND of local index and double-pointer index is true
     for(i=0; i<size; ++i)
            if (**flagPtr & i)
```

```
                x += *(ptr+i);
    return x;
}

data_t pointer_double(data_t pos, data_t x, data_t* flag)
{
    data_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    data_t* ptrFlag;
    data_t i;

    ptrFlag = flag;

   // Write x into index position pos
   if (pos >=0 & pos < 10)
     *(array+pos) = x;

    // Pass same index (as pos) as pointer to another function
    return sub(array, 10, &ptrFlag);
}
```

Example 4-32: **Double Pointers**

Arrays of pointers can also be synthesized, as shown in Example 2-33 where an array of pointers is used to store the start location of the 2nd dimension of a global array.

Note: The pointers in an array of pointers can only point to a scalar or to an array of scalars. They cannot point to other pointers.

```
#include "pointer_array.h"

data_t A[N][10];

data_t pointer_array(data_t B[N*10]) {
    data_t i,j;
    data_t sum1;

    // Array of pointers
    data_t* PtrA[N];

    // Store global array locations in temp pointer array
    for (i=0; i<N; ++i)
        PtrA[i] = &(A[i][0]);

    // Copy input array using pointers
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            *(PtrA[i]+j) = B[i*10 + j];

    // Sum input array
    sum1 = 0;
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            sum1 += *(PtrA[i] + j);

    return sum1;
}
```

Example 4-33: **Pointer Arrays**

Pointer casting is supported for synthesis if native C types are used. In Example 2-34, type `data_t (char)` is cast to type `'`.

```
#define N 1024

typedef int data_t;
typedef char dint_t;

data_t pointer_cast_native (data_t index,  data_t A[N]) {
     dint_t* ptr;
     data_t i =0, result = 0;
     ptr = (dint_t*)(&A[index]);

     // Sum from the indexed value as a different type
     for (i = 0; i < 4*(N/10); ++i) {
            result += *ptr;
            ptr+=1;
     }
     return result;
}
```

Example 4-34: **Pointer Casting with Native Types**

Pointer casting is not however supported between general types. For example, if a (struct) composite type of signed values is created, the pointer cannot be cast to assign unsigned values.

```
struct {
     short first;
     short second;
} pair;

// Not supported for synthesis
*(unsigned*)pair = -1U;
```

In such cases, the values must be assigned using the native type(s).

```
struct {
     short first;
     short second;
} pair;

// Assigned value
pair.first = -1U;
pair.second = -1U;
```

C Libraries

Vivado HLS provides a number of C libraries to help allow common hardware design constructs to be both easily modeled in C and synthesized to RTL. The following C libraries are discussed in this chapter:

- The `hls_math.h` library
- The `hls_video.h` library

The `hls_math.h` library provides synthesizable versions of the most commonly used functions in the standard C/C++ `math.h` library to be synthesized.

The `hls_video.h` library provides video data types and classes which allow video algorithms to be more easily captured in C.

HLS Math Library

The Vivado HLS library hls_math.h is used to provide extensive support for the synthesis of the floating point functions found in the standard C and C++ libraries, math.h and cmath.h respectively.

Table 2-3 lists the functions in math.h or cmath.h which will be synthesized. Some of these functions will be implemented using a floating point LogicCore. The others will be implemented as bit-approximate implementations using the hls_math.h library.

- A bit-approximate implementation may not provide the exact same accuracy as the standard operation. The accuracy is typically within 1 ULP (Unit of Last Place) over most operating ranges but may be as large as 100 ULP.
- A bit-approximate implementation may use a different underlying algorithm, than the C/C++ software version, to achieve the result.

Functions in the math.h or cmath.h libraries not listed in Table 2-3 cannot be synthesized: Vivado HLS will halt with an error.

The hls_math.h library is automatically called by Vivado HLS when synthesis is performed. It can be optionally included in the source. The difference between including the hls_math.h library and not is the difference in C simulation results: these differences are later.

- For the floating point C functions listed in Table 2-3, there is no double-precision version of the function. As such, the support is marked as Not Applicable in Table 2-3.
- The figures in the Accuracy column list the accuracy difference from the minimum to maximum difference over the entire range of the operator input values.

Table 4-3: Math.h Bit - Approximate Supported Functions

Function	Float	Double	Accuracy (ULP)	LogicCore
ceilf	Supported	Not Applicable	Exact	Not Supported
copysignf	Supported	Not Applicable	Exact	Not Supported
fabsf	Supported	Not Applicable	Exact	Not Supported
floorf	Supported	Not Applicable	Exact	Not Supported
logf	Supported	Not Applicable	1 to 5	Not Supported
cosf	Supported	Not Applicable	1 to 100	Not Supported
sinf	Supported	Not Applicable	1 to 100	Not Supported
abs	Supported	Supported	Exact	Not Supported
ceil	Supported	Supported	Exact	Not Supported
copysign	Supported	Supported	Exact	Not Supported
cos	Supported	Supported	2 for float. 5 for double	Not Supported
fabs	Supported	Supported	Exact	Not Supported
floor	Supported	Supported	Exact	Not Supported
fpclassify	Supported	Supported	Exact	Not Supported
isfinite	Supported	Supported	Exact	Not Supported
isinf	Supported	Supported	Exact	Not Supported
isnan	Supported	Supported	Exact	Not Supported
isnormal	Supported	Supported	Exact	Not Supported
log	Supported	Supported	1 for float, 16 for double	Not Supported
log10	Supported	Supported	1 for float, 16 for double	Not Supported
recip	Supported	Supported	Exact	Supported
round	Supported	Supported	Exact	Not Supported
rsqrt	Supported	Supported	Exact	Supported
signbit	Supported	Supported	Exact	Not Supported
sin	Supported	Supported	2 for float, 5 for double	Not Supported
sqrt	Supported	Supported	Exact	Supported
trunc	Supported	Supported	Exact	Not Supported

With respect to the following seven functions in Table 2-3.

- isinf
- isnan

- copysign
- fpclassify
- isfinite
- isnormal
- signbit

Depending on the C standard being used to compile the above functions, the following variations in results may be seen:

C90 mode: Only isinf, isnan, and copysign are usually provided by the system header files, and they operate on doubles. In particular, copysign will always return a double result - this may result in unexpected results after synthesis if it has to be returned to a float, since a double-to-float conversion block will be introduced into the hardware.

C99 mode (-std=c99): All seven functions are usually provided under the expectation that the system header files will redirect them to __isnan(double) and __isnan(float). Note that the usual GCC header files don't redirect isnormal but implement it in terms of fpclassify.

C++ using math.h: All seven are provided by the system header files, and they operate on doubles. In particular, copysign will always return a double result - this may result in unexpected results after synthesis if it has to be returned to a float, since a double-to-float conversion block will be introduced into the hardware.

C++, using cmath: Similar to C99 mode (-std=c99) except the system header files are usually different, and that the functions are properly overloaded for float(). Note that isnan(double) and isinf(double), copysign and copysignf are handled as built-ins, even when 'using namespace std;'

C++, using cmath and namespace std: No issues

Note: GCC usually implements isinf and copysign as built-ins and reduces isnan to a floating point comparison. I think we need to use -fno-builtin by default to avoid this behavior and get the best behavior.

It is recommended to use -std=c99 for C and -fno-builtin for C and C++ for best results.

Simulation Differences

The results obtained by simulation will vary depending on which C library is used for the C simulation and whether the C is compare to the RTL simulation is using the bit-approximate implementation. Table 2-4 shows the difference in simulation results.

In summary:

- If the math.h or cmath.h library is used, there will be a difference between the C and RTL results. (an example of this difference is shown below).

- If the hls_math.h library is used, there will be a difference between these results and those obtained using math.h or cmath.h.
- If math.h or cmath.h library is used, and file lib_hls.cpp is included, the results will be the same as if hls_math.h was used.

Table 4-4: Simulation Result Matrix

	math.h or cmath.h	Hls_math.h	math.h or cmath.h with lib_hls.cpp included.
C Validation	Results A	Results C	Results C
RTL Simulation	Results B	Results C	Results C

The following C++ Synthesis of math.h Functions example, shows a C++ design using the `sinf`, `cosf` and `sqrtf` functions, and highlights how Vivado HLS can synthesize the design but how the pre-synthesis and post-synthesis results will be different.

```
#include "cpp_math.h"

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

The header file `cpp_math.h` is shown in C++ Synthesis of math.h Functions header file example and shows how the `cmath` library is called (`math.h` is used in C), the standard namespace is used and the data type `data_t` is defined as a `float` type.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle);
```

Although the test bench shown in C++ Synthesis of math.h functions Test Bench example will validate the results in a pre-synthesis simulation, it will return an error after post-synthesis RTL simulation: this is discussed next.

```
#include "cpp_math.h"

int main() {
    ofstream result;
    data_t angle = 0.01;
    data_t output;
  int retval=0;
```

```
    result.open("result.dat");
    // Persistent manipulators
    result << right << fixed << setbase(10) << setprecision(15);

for (data_t i = 0; i <= 250; i++)
      {
        output = cpp_math(angle);

        result << setw(10) << i;
        result << setw(20) << angle;
        result << setw(20) << output;
        result << endl;

        angle = angle + .1;
      }
    result.close();

// Compare the results file with the golden results
  retval = system("diff --brief -w result.dat result.golden.dat");
  if (retval != 0) {
    printf("Test failed  !!!\n");
    retval=1;
  } else {
    printf("Test passed !\n");
   }

  // Return 0 if the test passes
  return retval;
}
```

The test bench performs a comparison of the output results, saved to file `result.dat`, with the expected data values in `result.golden.dat`. After synthesis, the RTL implementation of the `math.h` functions will be bit-approximate versions of the functions and the `results.dat` file output by the RTL simulation may therefore contain different results than those expected.

Vivado HLS executes the RTL simulation in the project sub-directory <SOLUTION>/sim/<HDL>, where SOLUTION is the name of the solution and HDL is the HDL type chosen for RTL simulation. For example, given the following project settings:

- Project is called `proj_cpp_math.prj`
- Solution is `solution1`
- RTL is simulated using `systemc`

the RTL simulation output will be saved in file in `proj_cpp_math.prj/solution1/sim/ systemc/results.dat`.

Figure 2-6 shows a comparison of the pre-synthesis `result.dat` file and the post-synthesis RTL `result.dat` file: the output value is shown in the 3rd column.

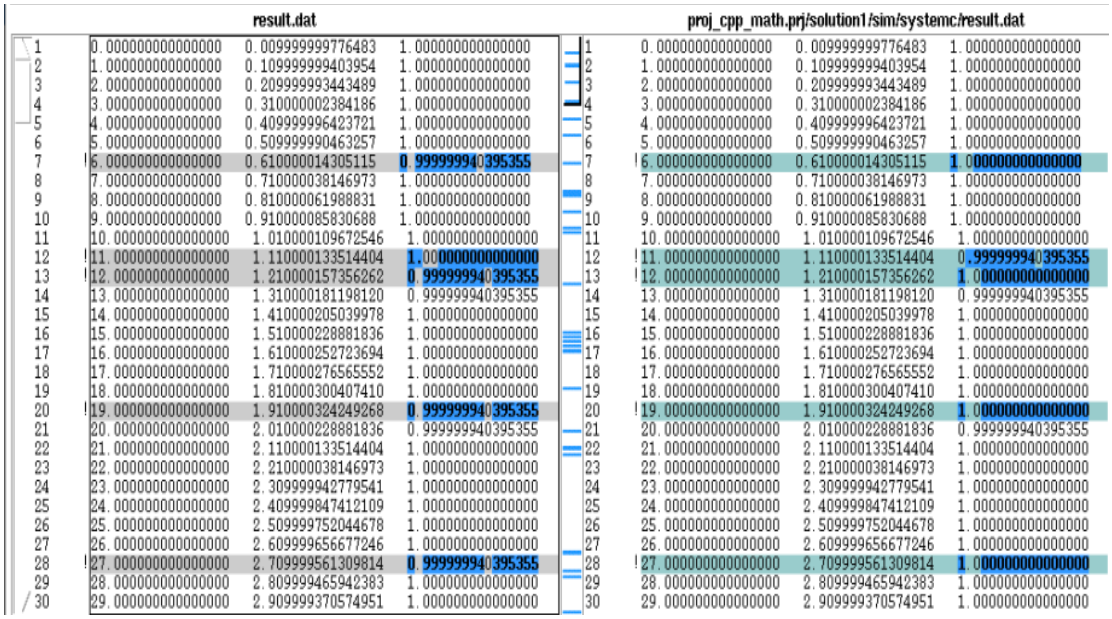

Figure 4-6: Pre-Synthesis and Post-Synthesis Simulation Differences

The results of pre-synthesis simulation and post-synthesis simulation differ, in this algorithm and test bench, by fractional amounts. The question is whether these fractional amounts are acceptable in the final RTL implementation.

The recommended flow for handling these differences relies on using a smart test bench which checks the results to ensure they lie within an acceptable error range.

- If using the math.h and cmath.h libraries (as in this case), verify the differences in accuracy are acceptable.

- Alternatively, convert the function to use hls_math.h and verify the differences between these results and those using math.h or cmath.h are acceptable.

Note: When working with floating point and double precision operations and functions, it is advisable to check the results using ranges. Using absolute comparisons will often lead to errors (even when comparing two sets of results calculated in a similar, but not identical manner, in the C code).

Common Synthesis Errors

The following are common use errors when synthesizing math functions. These are often, but not exclusively, caused by converting C functions to C++ in order to take advantage of synthesis for math functions.

If the C++ `cmath.h` header file is used, the floating point functions (`sinf`, `cosf`, etc) can be used and these will result in 32-bit operations in hardware. The `cmath.h` header file also

overloads the standard functions (`sin`, `cos`, etc) so that they can be used for float and double types.

If the C `math.h` library is used, the floating point functions (`sinf`, `cosf`, etc) are required in order to synthesize 32-bit floating point operations. All standard function calls (`sin`, `cos`, etc.) will result in doubles and 64-bit double-precision operations being synthesized.

Note: When converting C functions to C++ in order to take advantage of `math.h` support, ensure the new C++ code compiles correctly before synthesizing with Vivado HLS.

For example, if `sqrtf()` is used in the code with `math.h` it requires the following code `extern "C" float sqrtf(float);` added to C++ code to support it.

Also, follow the warnings on mixing double and float types, outlined in section Floats and Doubles, to avoid unnecessary hardware caused by type conversion.

HLS Video Library

The Vivado HLS video libraries require the use of the `hls_video.h` header file. All image and video processing specific video types and functions provided by Vivado HLS are provided with this header file.

When using Vivado HLS video libraries, the only additional usage requirement is that the design is written in C++ and uses the hls namespace, or the types and classes must use scoped naming.

```
#include <hls_video.h>

hls::rgb_8 video_data[1920][1080]
```

or

```
#include <hls_video.h>
using namespace hls;

rgb_8 video_data[1920][1080]
```

Data Types

The following data types are provided in the library. All data types currently support 8-bit data only.

Table 4-5: **Video Data Types**

Data Type Name	Field 0 (8 bits)	Field 1 (8 bits)	Field 2 (8 bits)	Field 3 (8 bits)
yuv422_8	Y	UV	Not Used	Not Used
yuv444_8	Y	U	V	Not Used

Table 4-5: **Video Data Types**

Data Type Name	Field 0 (8 bits)	Field 1 (8 bits)	Field 2 (8 bits)	Field 3 (8 bits)
rgb_8	G	B	R	Not Used
yuva422_8	Y	UV	a	Not Usedd
yuva444_8	Y	U	V	a
rgba_8	G	B	R	a
yuva420_8	Y	aUV	Not Used	Not Used
yuvd422_8	U	UV	D	Not Used
yuvd444_8	Y	U	V	D
rgbd_8	G	B	R	D
bayer_8	RGB	Not Used	Not Used	Not Used
luma_8	Y	Not Used	Not Used	Not Used

Once the `hls_video.h` library is included and the hls namespace defined, the data types listed in Example 2-5 can be freely used.

```
#include <hls_video.h>
using namespace hls;

rgb_8 video_data[1920][1080]
```

Memory Line Buffer

The `linebuffer` class is a C++ class which allows the user to easily declare and manage line buffers within their algorithmic code. This class provides all the methods required for instantiating and working with line buffers. The linebuffer class works with all data types.

The main features of the linebuffer class are

- Support for all data types through parameterization
- User defined number of rows and columns
- Automatic banking of rows into separate memory banks for increased memory bandwidth
- Provides all the methods for using and debugging line buffers in an algorithmic design

The linebuffer class has the following methods, explained below.

- print()
- shift_up();
- shift_down()
- insert_bottom()

- insert_top()
- getval(row,column)

In order to illustrate the usage of the linebuffer class, the following data set is assumed at the start of all examples.

Table 4-6: Data Set for Linebuffer Examples

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 2	1	2	3	4	5
Row 1	6	7	8	9	10
Row 0	11	12	13	14	15

Line Buffer Declaration

A line buffer can be instantiated in an algorithm by using the following data type:

```
hls::linebuffer<type, rows, columns>
```

One possible declaration for the line buffer holding the data in Table 2-6 is

```
hls::linebuffer<char,3,5> Buff_A;
```

Displaying Contents of the Line Buffer

The linebuffer class provides a print method to display the stored data in the line buffer. Since a line buffer can have a large number of columns, the print method displays all rows between a start and end column value.

For example,

```
Buff_A.print(1,3);
```

results in

```
Line 2:234
Line 1:789
Line 0:121314
```

Inserting and Shifting Data in a Line Buffer

The linebuffer class assumes that the data entering the block instantiating the line buffer is arranged in raster scan order. Each new data item will therefore be stored in a different column than the previous data item.

Inserting new values, while preserving a finite number of previous values in a column, requires a vertical shift between rows for a given column. After the shift is complete, a new data value can be inserted at either the top or the bottom of the column.

For example, inserting the value 100 to the top of column 2 of the line buffer set can be accomplished by using the following methods:

```
Buff_A.shift_down(2);
Buff_A.insert_top(100,2);
```

which will result in the new data set shown in Table 2-7.

Table 4-7: Data Set after Shift Down and Insert Top Classes Used

	Column 0	Column 1	Column 2	Column 3	Column 4
Line 2	1	2	100	4	5
Line 1	6	7	3	9	10
Line 0	11	12	8	14	15

Similarly, inserting the value 100 to the bottom of column 2 of the line buffer set in Table 2-8 can be accomplished by

```
Buff_A.shift_up(2);
Buff_A.insert_bottom(100,2);
```

which results in a new data set as shown in Table 2-8.

Table 4-8: Data Set after Shift Up and Insert Bottom Classes Used

	Column 0	Column 1	Column 2	Column 3	Column 4
Line 2	1	2	100	4	5
Line 1	6	7	3	9	10
Line 0	11	12	8	14	15

The shift and insert methods both require the column value on which to operate on.

Retrieving Data

All values stored by a linebuffer instance are available using the `getval(row,column)` method. This method returns the value of any location inside the line buffer. For example,

```
Value = Buff_A.getval(1,3);
```

results in variable `Value` being assigned the value 9.

Memory Window

The window C++ class allows the user to declare and manage 2 dimensional memory windows. The main features of this class are

- Support for all data types through parametrization
- User defined number of rows and columns

- Automatic partitioning into individual registers for maximum bandwidth
- Provides all the methods to use and debug memory windows in the context of an algorithm

The memory window class is supported by the following methods, explained below:

- print()
- shift_up();
- shift_down()
- shift_left()
- shift_right()
- insert(value,row,column)
- insert_bottom()
- insert_top()
- insert_left()
- insert_right()
- getval(row, column)

In order to illustrate the usage of the window class, the following data set is used at the start of all examples.

Table 4-9: Data Set for Memory Window Examples

	Column 0	Column 1	Column 2
Row 2	1	2	3
Row 1	6	7	8
Row 0	11	12	13

Window Declaration

A memory window can be instantiated in an algorithm using the following data type.

```
hls::window<type, rows, columns>
```

One possible declaration for a memory window holding the data.

```
hls::window<char,3,3> Buff_B;
```

Displaying Contents of the Memory Window

The window class provides a print method for displaying the contents of a memory window. By default, this method displays all values stored in the window. For example:

```
Buff_B.print();
```

results in

```
Window Size3x3
Col012
Row 2123
Row 1678
Row 0111213
```

For all window class instantiations, row 0 is assumed to be the bottom of the memory window.

Shifting Data in a Memory Window

The window class provides methods for moving data stored within the memory window up, down, left and right. Each shift operation will clear space in the memory window for new data.

Starting with the data set, these shifts have

```
Buff_B.shift_up();
Buff_B.print();
```

the following results.

```
Window Size3x3
Col0 1   2
Row 267  8
Row 1111213
Row 0New dataNew dataNew data
```

Similarly, starting with the data set in Table 2-9, these shifts have

```
Buff_B.shift_down();
Buff_B.print();
```

the following results.

```
Window Size3x3
Col 0   1   2
Row 2New dataNew dataNew data
Row 11 2   3
Row 06 7   8
```

And operations

```
Buff_B.shift_left();
Buff_B.print();
```

Will shift the data left and result in:

```
Window Size3x3
Col012
Row 223New data
```

```
Row 178New data
Row 01213New data
```

Finally,

```
Buff_B.shift_right();
Buff_B.print();
```

Will result in:

```
Window Size3x3
Col0 1 2
Row 2New data12
Row 1New data67
Row 0New data1112
```

Inserting and Retrieving Data from the Memory Window

The window class allows the user to insert and retrieve data from any location within the memory window. It also supports block insertion of data on the boundaries of the memory window.

Data can be inserted into any location of the memory window by `insert(value,row,column)`. For example, the value 100 can be placed into row 1, column 1 of the memory window by

```
Buff_B.insert(100,1,1);
Buff_B.print();
```

which according to the print method results in

```
Window Size3x3
Col012
Row 2123
Row 161008
Row 0111213
```

Block level insertion requires the user to provide an array of data elements to be inserted on a boundary. The methods provided by the window class are

- insert_bottom
- insert_top
- insert_left
- insert_right

For example, given the case where C is an array of 3 elements in which each element has the value of 50, inserting the value 50 across the bottom boundary of the memory window is achieved by

```
char C[3] = {50, 50, 50};
```

```
Buff_B.insert_bottom(C);
Buff_B.print();
```

results in

```
Window Size3x3
Col012
Row 2123
Row 161008
Row 0505050
```

The other edge insertion methods for the window class work in the same way as the `insert_bottom()` method.

Data can be retrieved from a memory window using `getval(row,column)`. For example,

```
A = Buff_B.getval(0,1);
```

results in

```
A = 50
```

Coding Styles for Modeling Hardware

C code is written to satisfy a number of different requirements: reuse, readability, performance. Until now, it is unlikely the C code was written to result in the most ideal hardware after High-Level Synthesis.

However, like the requirements for reuse, readability and performance, certain coding techniques or pre-defined constructs can be used to ensure the synthesis output results in more optimal hardware or to better model hardware in C for easier validation of the algorithm.

User Defined Registers in C++

In general, the only variables in a C function which are guaranteed to be implemented as registers are those proceeded by the `static` qualifier, those defined in the global scope (unless the global is exposed as a port) and arrays which are targeted to memory resources.

For the other variables, they may be implemented in a register, or they may not: it depends on the decisions made during of synthesis. Synthesis may determine that a particular variable should be registered over multiple cycles, or it may determine the variable can be used in the same cycle it's created and therefore is not required to be registered. Only the variable types mentioned above are guaranteed to be registers in the RTL.

The C++ function, "Reg", used in the following example is a useful technique to guarantee (or force) that a particular variable is a register in the final RTL design.

```
#include "foo.h"

template<class T>
T Reg(T in) {
#pragma AP INLINE off
#pragma AP INTERFACE port=return register
  return in;
}

int foo (int in1, int in2 ) {
  int tmp=in1*(0x0800-in2);
  // int out = tmp >> 10;
  int out = Reg(tmp >> 10);
  return( out );
}

int foo_top(int inA, int inB) {

  int res1 = foo(inA, inB);
  return(res1);
}
```

The important aspects of this code are:

- Function "Reg" is created and has its output, the function `return` value, registered.
  - Registering function arguments (the RTL ports) is the only allowed use of the interface directive on sub-functions. The interface directive cannot be used to specify the IO protocol of a sub-function argument.
- Function "Reg" has inlining disabled in case Vivado HLS decides to automatically inline this small function.
  - Inlining function "Reg" negate the register operation performed on the function, since it would no longer exist as a separate function if it is inlined.
- The output of the multiplier and shift operation in sub-function "foo" is used as the input to the "Reg" function.
  - This guarantees the result of this operation is registered in the RTL output, since the function "Reg" has a registered output.
- The original source code for this variable is shown commented out. This is shown how easy it is to apply this technique

In general, Vivado HLS will determine which variables should be registered in the final RTL, but this coding technique can be used to force particular variables to be registered without modifying the functionality of the code.

Mapping Directly into SRL resources

Many C algorithms sequentially shift data through arrays: add a new value to the start of the array, shift the existing data through array and drop the oldest data value. This operation is implemented in hardware as a shift-register.

This most common way to implement a shift-register from C into hardware is to completely partition the array into individual elements, and allow the data dependencies between the elements in the RTL to imply a shift-register.

Logic synthesis will typically implement the RTL shift-register into a Xilinx SRL resource, which efficiently implements shift-registers. The problem is that sometimes logic synthesis does not implement the RTL shift-register using an SRL component:

- When data is accessed in the middle of the shift-register, logic synthesis cannot directly infer an SRL.
- Sometimes, even when the RTL is ideal, logic synthesis may implement the shift-resister in flip-flops, due to other factors. (Logic synthesis is also a complex process).

Vivado HLS provides a C++ class, `ap_shift_reg`, which ensures the shift-register defined in the C code, is always implemented using an SRL resource. The `ap_shift_reg` class has two methods to perform the various read and write accesses supported by an SRL component.

Read From the Shifter

The `read` method allows a specified location to be read from the shifter register.

```
  // Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 2 of Sreg into var1
var1 = Sreg.read(2);
```

Read, Write and Shift Data

A `shift` method allows a read, write and shift operation to be performed.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
```

```
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 3 of Sreg into var1
// THEN shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3);
```

Read, Write and Enable-Shift

The `shift` method also supports an enabled input, allowing the shift process to be controlled/enable by a variable.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;

// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3,En);
```

When using the `ap_shift_reg` class, Vivado HLS will create a unique RTL component for each shifter. When logic synthesis is performed, this component is synthesized into an SRL resource.

Designing with Streaming Data

Streaming data is a type of data transfer in which data samples are sent in sequential order starting from the first sample: streaming requires no address management.

Modeling designs which use streaming data can be difficult in C. As discussed in the section Multi-Access Pointer Interfaces: Streaming Data, the approach of using pointers to perform multiple read and/or write accesses can introduce problems, since there are implications for the type qualifier and how the test bench is constructed.

Vivado HLS provides a C++ template class, `hls::stream<>`, for modeling streaming data structures. The streams implemented with the `hls::stream<>` class have the following attributes.

This section shows how the `hls::stream<>` class can be used to more easily model designs with streaming data. The topics in this section provide:

- An overview of modeling with streams and the RTL implementation of streams.
- Rules for global stream variables.
- How to use streams.
- Blocking reads and writes.
- Non-Blocking Reads and writes.
- Controlling the FIFO depth.

C Modeling and RTL Implementation

Streams are modeled as infinite queue in software (and in the test bench during RTL co-simulation). There is no need to specify any depth in order to simulate streams in C. Streams can be used inside functions and on the interface to functions. Internal streams may be passed as function parameters.

Note: Streams can only be used in C++ based designs. Each hls::stream<> object must be written to by a single process and read by a single process, e.g. in a DATAFLOW design each stream may only have one producer and one consumer process.

In the RTL streams are implemented as either a FIFO or full handshake interface port. The default interface port is an `ap_fifo` port. This port can optionally be implemented as an `ap_hs` port on the top-level interface: Internal stream will only be implemented as FIFO interfaces.

FIFOs created by streams are by default implemented with a depth of 1, implemented as a 2 element register-based FIFO. The depth of the FIFO optionally can be using the STREAM directive.

Global and Local Streams

Streams may be defined either locally or globally. Local streams will always be implemented as internal FIFOs. Global streams may be implemented as FIFOs or ports:

- Globally defined streams that are only read from, or only written to, will be inferred as external ports of the top-level RTL block.
- Globally defined streams that are both read from and written to (in the hierarchy below the top-level function) will be implemented as internal FIFOs.

Global variables may be interpreted as either internal FIFOs or external ports (top-level RTL ports) to an HLS design, depending on usage, tool options and any C++ qualifier applied at declaration.

The rules for determining whether or not a globally declared `hls::stream<>` object will be created as an internal FIFO or external port interface are as follows.

- If the 'static' qualifier is applied at definition, the stream may only be accessed by functions defined in the same source file as the definition (to conform to C/C++ semantics) and will be considered internal and implemented as a FIFO.
- If the config_interface command is in effect and has the -expose_global option specified **all** non-static global `hls::stream<>` objects (as well as any other non-static global variables of other types) will implement as ports in the top-level RTL.

Otherwise, any `hls::stream<>` object which is both read from and written to within the design as an internal FIFO; Any that cannot be resolved to have internal linkage will be exposed as top-level ports.

Using Streams

To use `hls::stream<>` objects the header file `hls_stream.h` must be included. This file is available in the Vivado HLS include directory in the installation area (this directory is auto-searched during simulation and synthesis and does not need to added to the search path).

Streaming data objects are defined by specifying the type and variable name. In this example, a 128-bit unsigned integer type is defined and used to create a stream variable called `my_wide_stream`.

```
#include <ap_int.h>
#include <hls_stream.h>

typedef ap_uint<128> uint128_t; // 128-bit user defined type
hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

Given that a stream is specified as hls::stream<T>, the type T may be any C++ native data type, an HLS arbitrary precision type (e.g. ap_int<>, ap_ufixed<>, etc) or a user defined (typedef) structure type.

Note: General user defined classes (or structures) that contain methods (member functions) should not be used as the type (T) for a stream variable.

Streams must use scoped naming, as in the above example, or include the hls namespace at the file-scope with a "using namespace hls;" statement. If a namespace is used, the above example can be re-written as:

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;

typedef ap_uint<128> uint128_t; // 128-bit user defined type
stream<uint128_t> my_wide_stream; // hls:: no longer required
```

It is highly recommended to use pass-by-reference only, e.g. foo(hls::stream<char> &foo, ...), as this allows method selection via the . operator rather than the > operator.

Blocking and non-blocking access methods are supported.

- Non-blocking accesses can only be implemented as FIFO interfaces.
- Streaming ports, implemented as `ap_fifo` ports, and which will be defined with to an AXI4Stream resource, must not use non-blocking accesses.

Blocking Reads and Writes

The basic accesses to an hls::stream<> object are blocking reads and writes, which are accomplished via class methods (member functions). These methods will stall (block) execution if a read is attempted on an empty stream FIFO, a write to a full stream FIFO or until a full handshake is accomplished for a stream mapped to an ap_hs interface protocol.

Blocking Write Methods

In this example, the value of variable `src_var` is pushed into the stream.

```
// Usage of void write(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream.write(src_var);
```

The << operator is overloaded such that it may be used in a similar fashion to the stream insertion operators for C++ stream (e.g. iostreams, filestreams, etc). The `hls::stream<>` object to be written to is supplied as the left-hand side argument and the value to be written as the right-hand side.

```
// Usage of void operator << (T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream << src_var;
```

Blocking Read Methods

This method reads from the head of the stream and assigns the values to the variable dst_var.

```
// Usage of void read(T &rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream.read(dst_var);
```

Alternatively, the next object in the stream can be read by simply assigning (using =, +=, etc) the stream to an object on the left-hand side:

```
// Usage of T read(void)

hls::stream<int> my_stream;

int dst_var = my_stream.read();
```

The '>>' operator is overloaded to allow use similar to the stream extraction operator for C++ stream (e.g. iostreams, filestreams, etc). The hls::stream is supplied as the LHS argument and the destination variable the RHS.

```
// Usage of void operator >> (T & rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream >> dst_var;
```

Non-Blocking Reads and Writes

Non-blocking write and read methods are also provided. These allow execution to continue even when a read is attempted on an empty stream FIFO or a write to a full stream FIFO.

These methods return a Boolean value indicating the status of the access (true if successful, false otherwise). Additionally methods are provided for testing the status of an `hls::stream<>` stream.

Note: None of the methods detailed for non-blocking accesses may be used on an hls::stream<> interface for which the ap_hs protocol has been selected.

During C simulation, streams have an infinite size. It is therefore not possible to validate with C simulation if the stream will be full: these methods can only be verified during RTL simulation when the FIFO sizes are defined (either the default size of 1 or an arbitrary size defined with the STREAM directive).

Non-blocking Writes

This method attempts to push variable `src_var` into the stream `my_stream`, returning a boolean `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of void write_nb(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

if (my_stream.write_nb(src_var)) {
  // Perform standard operations
```

```
  ...
} else {
  // Write did not occur
  return;
}
```

Fullness Test

bool **full**(void)

This method returns true, if and only if the hls::stream<> object is full.

```
// Usage of bool full(void)

hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;

stream_full = my_stream.full();
```

Non-Blocking Read

bool **read_nb**(T & rdata)

This method attempts to read a value from the stream, returning `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of void read_nb(const T & wdata)

hls::stream<int> my_stream;
int dst_var;

if (my_stream.read_nb(dst_var)) {
  // Perform standard operations
  ...
} else {
  // Read did not occur
  return;
}
```

Emptiness Test

bool **empty**(void)

This method returns true if the hls::stream<> is empty.

```
// Usage of bool empty(void)

hls::stream<int> my_stream;
int dst_var;
bool stream_empty;

fifo_empty = my_stream.empty();
```

The following example shows how a combination of non-blocking accesses and full/empty tests can be used to provide error handling functionality for cases when the RTL FIFOs are full or empty:

```
#include "hls_stream.h"
using namespace hls;

typedef struct {
   short    data;
   bool     valid;
   bool     invert;
} input_interface;

bool invert(stream<input_interface>& in_data_1,
            stream<input_interface>& in_data_2,
            stream<short>& output
  ) {
  input_interface in;
  bool full_n;

// Read an input value or return
  if (!in_data_1.read_nb(in))
      if (!in_data_2.read_nb(in))
          return false;

// If the valid data is written, return not-full (full_n) as true
  if (in.valid) {
    if (in.invert)
      full_n = output.write_nb(~in.data);
    else
      full_n = output.write_nb(in.data);
  }
  return full_n;
}
```

A complete design example using streams is provided in the Vivado HLS examples section: `Help > Welcome > Examples > Design > hls_stream`.

Controlling the RTL FIFO Depth

For most designs using streaming data, the default RTL FIFO depth of 1 is sufficient: streaming data is generally processed one sample at a time.

For multi-rate designs where the implementation may require a FIFO with a depth greater than 1, you must determine, and set using the STREAM directive, the depth necessary for the RTL simulation to complete. If the FIFO depth is insufficient, the symptom will be that RTL co-simulation stalls.

Stream objects cannot be viewed in the GUI directives pane. As such, the STREAM directive cannot be applied directly using the GUI directives pane, however, by right-clicking on the function in which an `hls::stream<>` object is declared (or is used or exists in the

argument list) the STREAM directive may be selected and the 'variable' field may be populated manually with name of the stream variable.

An alternative is to specify the STREAM directive manually in the directives.tcl file or added as a pragma in source.

RTL Co-Simulation Support

At present, the following scenarios are not supported by the Vivado HLS RTL co-simulation feature (they are supported for synthesis):

Arrays of `hls::stream<>` in the top-level interface.

```
void dut_top(uint16_t odata[N], hls::stream<uint8_t>chan[4]) { … }
```

Structures or classes containing `hls::stream<>` members in the top-level interface.

```
typedef struct {
   hls::stream<uint8_t> a;
   hls::stream<uint16_t> b;
} strm_strct_t;

void dut_top(strm_strct_t indata, strm_strct_t outdata) { … }
```

These restrictions apply to both top-level function arguments and globally declared objects. If arrays or structs of streams are used for synthesis, the design must be verified using an external RTL simulator and user created HDL test bench. There are no such restrictions on `hls::stream<>` objects with strictly internal linkage.

C Arbitrary Precision Integer Types

The native data types in C are on 8-bit boundaries (8, 16, 32 and 64 bits). RTL signals and operations however support arbitrary bit-lengths. Vivado HLS provides arbitrary precision data types for C to allow variables and operations in the C code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit etc. up to 1024 bits.

Note: Vivado HLS also provides arbitrary precision data types in C++ and supports the arbitrary precision data types which are part of SystemC. These types are discussed in the respective C++ and SystemC coding.

There are two primary advantages of arbitrary precision data types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can be used to specify that exactly 17-bit are used in the calculation.
  - Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP48 components.

- Accurate C simulation/analysis: Arbitrary precision data types in the C code allows the C simulation to be executed using accurate bit-widths and for the C simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The remainder of this section explains how to use arbitrary precision types and reviews issues where care should be taken. A detailed description of arbitrary precision types is provided in a reference section at the end of this document (C Arbitrary Precision Types) and includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 64-bit).
- A description of Vivado HLS helper functions, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift values, results in a shift in the opposite direction).

Using Arbitrary Precision Types with C

For the C language, the header file `ap_cint.h` defines the arbitrary precision integer data types `[u]int#W`. For example, `int8` represents an 8-bit signed integer data type and uint234 represents a 234-bit unsigned integer type.

The `ap_cint.h` file is located in the directory `$HLS_ROOT/include`, where `$HLS_ROOT` is the Vivado HLS installation directory.

The code shown in Example 2-35, is a repeat of the code shown in the earlier example on basic arithmetic (Example 2-22). In both examples the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t` etc.

```
#include "apint_arith.h"

void apint_arith(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
                 out_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
  ) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;

}
```

Example 4-35: **Basic Arithmetic Revisited**

The real difference between the two examples is in how the data types are defined. To use arbitrary precision integer data types in a C function:

- Add header file `ap_cint.h` to the source code.

- Change the native C types to arbitrary precision types `intN` or `uintN`, where `N` is a bit-size from 1 to 1024.

The data types are defined in the header `apint_arith.h` as shown in Example 2-36. Compared with Example 2-22, the input data types have simply been reduced to represent the maximum size of the real input data (e.g., 8-bit input `inA` is reduced to 6-bit input). The output types, however, have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

```
#include <stdio.h>
#include "ap_cint.h"

// Previous data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;

typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;

void apint_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);
```

Example 4-36: **Basic Arithmetic APINT Types**

If Example 2-36 is synthesized it will result in a design which is functionally identical to Example 2-22 (given data in the range specified by Example 2-36). The final RTL design, however, is smaller in area and has a faster clock speed (smaller bit-widths result in reduced logic).

However, before synthesis, the function must be compiled and validated.

Validating Arbitrary Precision Types in C

To create arbitrary precision types, attributes are added to define the bit-sizes in file `ap_cint.h`. Standard C compilers such as `gcc` will compile the attributes used in the header file, but they do know what the attributes mean. The final executable created by standard C compilers will issue messages such as the following:

```
$HLS_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute directive
ignored
```

It will then proceed to use native C data types for the simulation. This results in computations which do not reflect the bit-accurate behavior of the code. For example, a 3-bit integer value with binary representation 100 will be treated as having a decimal value 4 and not -4.

Vivado HLS includes a compiler, `apcc`, which overcomes this limitation and allows the function to be compiled and simulated in a bit-accurate manner.

IMPORTANT: *When bit-accurate types are in C, the design must be compiled and simulated using the `apcc` compiler.*

The `apcc` compiler can be enabled in the project setting using menu **Project > Project Settings > Simulation** and selecting **Use APCC Compiler** as shown in Figure 2-7.

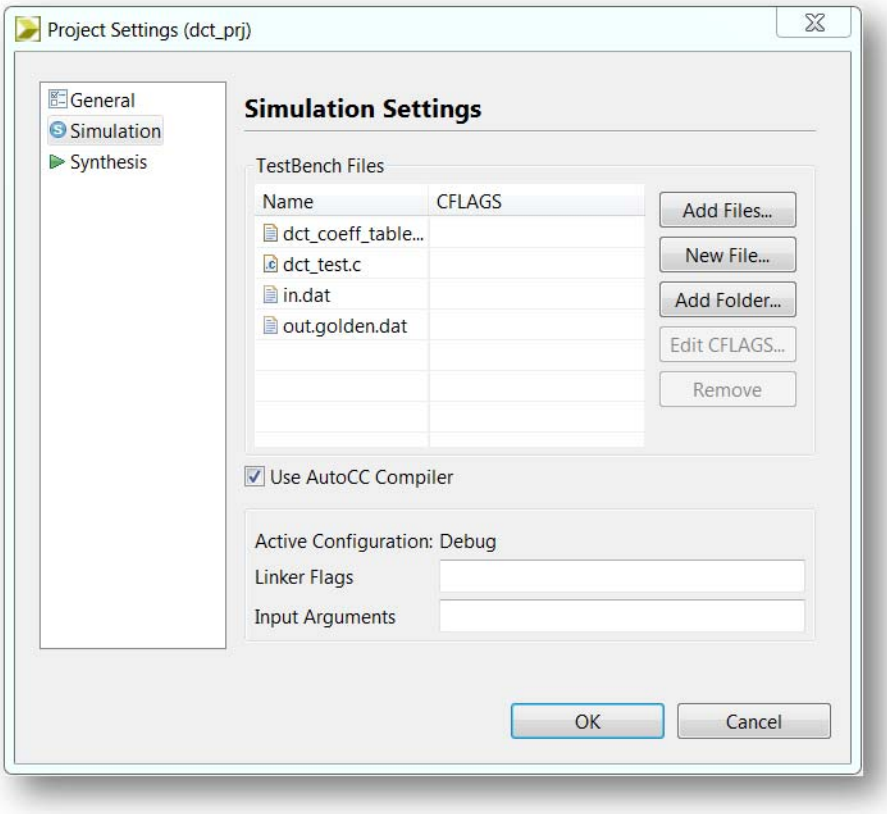

Figure 4-7: **Enabling the APCC Compiler**

If compiling at the command prompt, the `apcc` compiler should be used at the shell prompt: it is command line compatible with `gcc` and will process the arbitrary precision arithmetic correctly (respecting the boundaries imposed by the bit-width information).

When apcc is used, the Vivado HLS header files are automatically included (no need to use -I$HLS_ROOT/include) and the design will simulate with the correct bit-accurate behavior.

```
$ apcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

In summary, when using arbitrary precision types in C, compile using the apcc compiler:

- Select the **Use APCC Compiler** option in the GUI.
- Use apcc in place of gcc at the command prompt.

For functions specified using C++ or SystemC there are no such limitations when using arbitrary precision types. (An alternative may be to change the file name exertions to .cpp, use C++ arbitrary precision types, and compile/simulate using a C++ compiler).

Debugging Arbitrary Precision Types in C

Note: When apcc is used to compile C code the design can no longer be analyzed in the Vivado HLS C debugger: this is a side-effect of using arbitrary procession type in C code.

If there is a requirement to debug the design, the following methodology is recommended:

- If the operation of the algorithm requires analysis in the debugger, use native C types (int, char, short, etc.) and open the code in the debugger to verify correct operation of the algorithm.
  - This is easily performed when all types are defined in the header file, allowing the data types throughout to be changed in one location.
- If the operation of the algorithm is known to be correct and it is simply the case that the bit accurate nature of the arbitrary precision types must be analyzed, execute a C simulation and use the printf and/or fprintf functions to output the data values for analysis.

Integer Promotion Issues

Care should be taken when the result of arbitrary precision operations crosses the native 8, 16, 32 and 64-bit boundaries. In the following example, the intent is that two 18-bit values are multiplied and the result stored in a 36-bit number:

```
#include "ap_cint.h"

int18  a,b;
int36  tmp;

tmp = a * b;
```

However, what happens in this example is that integer promotion occurs and the result in not what is expected.

In integer promotion, the C compiler promotes the result of the multiplication operator from 18-bit, to the next native bit size (32-bit) and then assigns the result to the 36-bit variable `tmp`. This results in the behavior and incorrect result shown in Figure 2-8.

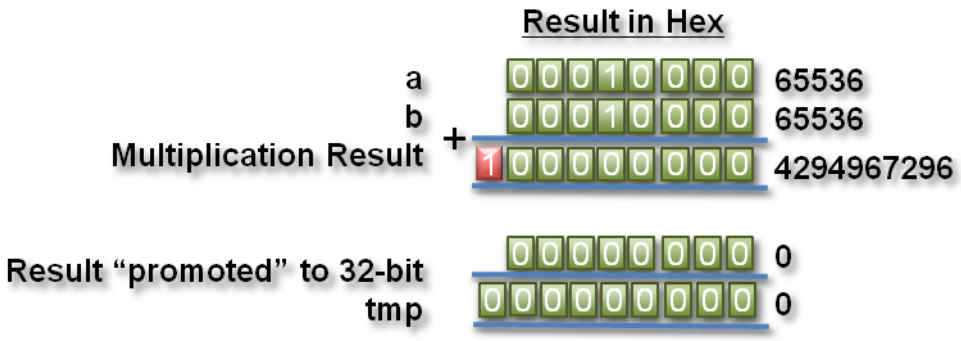

Figure 4-8: **Integer Promotion**

Since Vivado HLS will produce the same results as C simulation, Vivado HLS will create hardware where a 32-bit multiplier result is sign-extended to a 36-bit result.

The solution to the integer promotion issue is to cast operator inputs to the output size. Figure 2-37 shows where the inputs to the multiplier are cast to 36-bit value before the multiplication. This results in the correct (expected) results during C simulation and the expected 36-bit multiplication in the RTL.

```
#include "ap_cint.h"

typedef int18 din_t;
typedef int36 dout_t;

dout_t apint_promotion(din_t a,din_t b) {
  dout_t  tmp;

  tmp = (dout_t)a * (dout_t)b;
  return tmp;
}
```

Example 4-37: **Cast to avoid Integer Promotion**

Casting to avoid integer promotion issue is only required when the result of an operation is greater than the next native boundary (8, 16, 32 or 64). This behavior is more typical with multipliers than with addition and subtraction operations.

Integer promotion issues are not present when using C++ or SystemC arbitrary precision types.

Functions

The top-level function becomes the top-level of the RTL design after synthesis and sub-functions are synthesized into blocks in the RTL design.

IMPORTANT: The top-level function cannot be a static function.

After synthesis, each function in the design will have its own synthesis report and RTL HDL file (Verilog, VHDL and SystemC). Sub-functions can optionally be inlined to merge their logic with the logic of the surrounding function. Inlining functions can result in better optimizations but can also increase run time, since more logic and more possibilities have to be kept in memory and analyzed. Vivado HLS may perform automatic inlining of small functions (which can be disabled by setting the `inline` directive to `off` for that function). If a function is inlined there will be no report or separate RTL file for that function: the logic and loops are merged with the function above it in the hierarchy.

The primary impact of a coding style on functions is on the function arguments and interface.

If the arguments to a function are sized accurately, Vivado HLS can propagate this information through the design and there is no need to create arbitrary precision types for every variable. In the following example, two integers are multiplied, but only the bottom 24-bits are used for the result.

```
#include "ap_cint.h"

int24 foo(int x, int y) {
     int tmp;

     tmp = (x * y);
     return tmp
}
```

When this code is synthesized the result will be a 32-bit multiplier with the output truncated to 24-bit.

If, however, the inputs are correctly sized to 12-bit types (int12) as shown in Example 2-38, the final RTL will use a 24-bit multiplier.

```
#include "ap_cint.h"
typedef int12 din_t;
typedef int24 dout_t;

dout_t func_sized(din_t x, din_t y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

Example 4-38: **Sizing Function Arguments**

Using arbitrary precision types for the two function inputs is enough to ensure Vivado HLS creates a design using a 24-bit multiplier: the 12-bit types are propagated through the design. It is recommended to correctly size the arguments of all functions in the hierarchy.

In general, when variables are driven directly from the function interface, especially from the top-level function interface, they can prevent some optimizations from taking place. A typical case of this is when an input is used as the upper limit for a loop index.

Loops

Loops provide a very intuitive and concise way of capturing the behavior of an algorithm and are used often in C code. Loops are very well supported by synthesis: loops can be pipelined, unrolled, partially unrolled, merged and flattened.

The optimizations unroll, partially unroll, flatten and merge effectively make changes to the loop structure, as if the code was changed: these optimizations ensure limited coding changes are required when optimizing loops. There are some optimizations however which can only be applied in certain conditions and some coding changes may be required to allow them.

RECOMMENDED: *Avoid use of global variables for loop index variables, as this can inhibit some optimizations.*

Variable Loop Bounds

Some of the optimizations Vivado HLS can apply are prevented when the loop has variable bounds. In Example 2-39 shown below, the loop bounds are determined by variable width, which is driven from a top-level input. In this case the loop is considered to have variables bounds, since Vivado HLS cannot know when the loop will complete.

```
#include "ap_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t code028(din_t A[N], dsel_t width) {

      dout_t out_accum=0;
      dsel_t x;

      LOOP_X:for (x=0;x<width; x++) {
            out_accum += A[x];
      }

      return out_accum;
}
```

Example 4-39: **Variable Loop Bounds**

Trying to optimize the design in Example 2-39 will uncover the problems which are introduced by variable loop bounds.

The first issue with variable loop bounds is that they prevent Vivado HLS from determining the latency of the loop. Vivado HLS can determine the latency to complete one iteration of the loop, but because it cannot statically determine the exact value of variable width, it does not know how many iteration are performed and thus cannot report the loop latency (the number of cycles to completely execute every iteration of the loop).

Where variable loop bounds are present, Vivado HLS will report the latency as a question mark (?) instead of using exact values. The following shows the result after synthesis of Example 2-39.

```
+ Summary of overall latency (clock cycles):
     * Best-case latency:    ?
     * Average-case latency: ?
     * Worst-case latency:   ?
+ Summary of loop latency (clock cycles):
     + LOOP_X:
            * Trip count: ?
            * Latency:    ?
```

The first problem with variable loop bounds is therefore that the performance of the design is unknown.

To overcome this problem Vivado HLS provides the `tripcount` directive. The `tripcount` directive allows a minimum, average and/or maximum `tripcount` to be specified for the loop: the tripcount is the number of loop iterations. If a maximum tripcount of 32 is applied to `LOOP_X` in Example 2-39, the report is updated to the following:

```
+ Summary of overall latency (clock cycles):
     * Best-case latency:    2
     * Average-case latency: 18
     * Worst-case latency:   34
+ Summary of loop latency (clock cycles):
     + LOOP_X:
            * Trip count: 0 ~ 32
            * Latency:    0 ~ 32
```

Note: The values provided by the user for the tripcount directives are used only for reporting and **have impact** on synthesis. The tripcount value simply allows Vivado HLS to report number in the report. This allows the user to see the effect of optimizations: solutions can now be complared.

Tripcount directives have no impact on the results of synthesis, only reporting.

The next steps in optimizing Example 2-39 for high throughput would be:

- Unroll the loop and allow the accumulations to occur in parallel.
- Partition the array input, or the parallel accumulations will be limited, by a single memory port.

If these optimizations are applied, the output from Vivado HLS will highlight the biggest problem with variable bound loops:

```
@W [XFORM-503] Cannot unroll loop 'LOOP_X' in function 'code028': cannot completely
unroll a loop with a variable trip count.
```

Since variable bounds loops cannot be unrolled, they not only prevent the unroll directive being applied, they also prevent pipelining of the levels above the loop.

IMPORTANT: *When a loop or function is pipelined, Vivado HLS will unroll all loops in the hierarchy below the function or loop. If there is a loop with variable bounds in this hierarchy it will prevent pipelining.*

When a loop or function is pipelined, Vivado HLS will unroll all loops in the hierarchy below the function or loop. If there is a loop with variable bounds in this hierarchy it will prevent pipelining.

The solution to loops with variable bounds is to make the number of loop iteration a fixed value with conditional executions inside the loop. The code from Example 2-39 can be re-written as shown in Example 2-40. Here, the loop bounds are explicitly set to the maximum value of variable width and the loop body is conditionally executed.

```
#include "ap_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t loop_max_bounds(din_t A[N], dsel_t width) {

      dout_t out_accum=0;
      dsel_t x;

      LOOP_X:for (x=0;x<N-1; x++) {
            if (x<width) {
                  out_accum += A[x];
            }
      }

      return out_accum;
}
```

Example 4-40: **Variable Loop Bounds Re-Written**

The for-loop (`LOOP_X`) in Example 2-40 can be unrolled: the loop has fixed upper bounds and Vivado HLS knows how much hardware to create. There will be `N (32)` copies of the loop body in the RTL design, each copy of the loop body will have conditional logic associated with it and be executed depending on the value of variable width.

Loop Pipelining

When pipelining loops, the most optimum balance between area and performance is typically found by pipelining the inner most loop. This is also results in the fastest run time. The code in Example 2-41 can be used to demonstrate the trade-offs when pipelining loops and functions.

```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {

     int i,j;
            static dout_t acc;

            LOOP_I:for(i=0; i < 20; i++){
                   LOOP_J: for(j=0; j < 20; j++){
                          acc += A[i] * j;
                   }
            }

     return acc;
}
```

Example 4-41: **Loop Pipeline**

If the inner-most (LOOP_J) is pipelined, there will be one copy of LOOP_J in hardware, (a single multiplier) and Vivado HLS will use the outer-loop (LOOP_I) to simply feed LOOP_J with new data. Only 1 multiplier operation and 1 array access need to be scheduled, then the loop iterations can be scheduled as single loop-body entity (20x20 loop iterations).

Note: When a loop or function is pipelined, any loop in the hierarchy below the loop or function being pipelined must be unrolled.

If the outer-loop (LOOP_I) is pipelined, inner-loop (LOOP_J) will be unrolled creating 20 copies of the loop body: 20 multipliers and 20 array accesses must now be scheduled. Then each iteration of LOOP_I can be scheduled as a single entity.

If the top-level function is pipelined, both loops must be unrolled: 400 multipliers and 400 arrays accessed must now be scheduled. It is very unlikely Vivado HLS will produce a design with 400 multiplications since in most designs data dependencies often prevent maximal parallelism, for example, in this case, even if a dual-port RAM is used for A[N] the design can only access two values of A[N] in any clock cycle.

The concept to appreciate when selecting at which level of the hierarchy to pipeline it to understand that pipelining the inner-most loop will give the smallest hardware with generally acceptable throughput for most applications. Pipelining the upper-levels of the hierarchy will unroll all sub-loops and can create many more operations to schedule (which could impact run time and memory capacity) but will typically give the highest performance design in terms of throughput and latency.

To summarize the above options:

- **Pipeline `LOOP_J`**: Latency will be approximately 400 cycles (20x20) and will require less than 100 LUTs and registers (the IO control and FSM are always present).
- **Pipeline `LOOP_I`**: Latency will be approximately 20 cycles but will require a few hundred LUTs and registers. About 20 times the logic as first option, minus any logic optimizations which can be made.
- **Pipeline function `loop_pipeline`**: Latency will be approximately 10 (20 dual-port accesses) but will require thousands of LUTs and registers (about 400 times the logic of the first option minus any optimizations which can be made).

Imperfect Nested Loops

When the inner-loop of a loop hierarchy is pipelined, Vivado HLS automatically flattens the nested loops, to reduce latency and improve overall throughput by removing any cycles caused by loop transitioning (the checks performed on the loop index when entering and exiting loops). Such checks can result in a clock delay when transitioning from one loop to the next (entry and/or exit). In Example 2-41, pipelining the inner-most loop would result in the following message from Vivado HLS.

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_pipeline'.
```

Nested loops can only be flattened if the loops are perfect or semi-perfect.

- Perfect Loops
  - Only the inner most loop has body (contents).
  - There is no logic specified between the loop statements.
  - The loop bounds are constant.
- Semi-perfect Loops
  - Only the inner most loop has body (contents)
  - There is no logic specified between the loop statements.
  - The outer most loop bound can be variable.

Example 2-42 shows a case where the loop nest is imperfect.

```
#include "loop_imperfect.h"

void loop_imperfect(din_t A[N], dout_t B[N]) {

     int i,j;
           dint_t acc;

           LOOP_I:for(i=0; i < 20; i++){
                 acc = 0;
                 LOOP_J: for(j=0; j < 20; j++){
                              acc += A[i] * j;
                 }
                 B[i] = acc / 20;
```

```
        }
}
```

Example 4-42: **Imperfect Nested Loops**

The assignment to `acc` and array `B[N]` inside `LOOP_I`, but outside `LOOP_J`, prevent the loops from being flattened. If `LOOP_J` in Example 2-42 is pipelined, the synthesis report will show the following:

```
+ Summary of loop latency (clock cycles):
    + LOOP_I:
        * Trip count: 20
        * Latency:    480
        + LOOP_J:
            * Trip count:     20
            * Latency:        21
            * Pipeline II:    1
            * Pipeline depth: 2
```

- The pipeline depth shows it takes 2 clocks to execute one iteration of `LOOP_J` (this will vary with the device technology and clock period).
- A new iteration can begin each clock cycle: Pipeline II is 1 (II is the Initiation Interval: cycles between each new execution of the loop body).
- It takes 2 cycles for the first iteration to output a result. Due to pipelining each subsequent iteration executes in parallel with the previous one and outputs a value after 1 clock. The total latency of the loop is 2 plus 1 for each of the remaining 19 iterations: 21.
- `LOOP_I`, requires 480 clock cycles to perform 20 iterations, thus each iteration of `LOOP_I` is 24 clocks cycles: this means there are 3 cycles of overhead to enter and exit `LOOP_J` (24 – 21 = 3).

Imperfect loop nests, or the inability to flatten loop them, results in additional clock cycles to enter and exit the loops. The code in Example 2-42 can be re-written to make the nested loops perfect and allow them to be flattened.

Example 2-43 shows how conditionals can be added to loop `LOOP_J` to provide the same functionality as Example 2-42 but allow the loops to be flattened.

```
#include "loop_perfect.h"

void loop_perfect(din_t A[N], dout_t B[N]) {

    int i,j;
        dint_t acc;

        LOOP_I:for(i=0; i < 20; i++){
            LOOP_J: for(j=0; j < 20; j++){
                if(j==0) acc = 0;
                acc += A[i] * j;
                if(j==19) B[i] = acc / 20;
```

```
                    }
                }

}
```

Example 4-43: **Perfect Nested Loops**

When Example 2-43 is synthesized, the loops are flattened:

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_perfect'.
```

And the synthesis report shows an improvement in latency.

```
+ Summary of loop latency (clock cycles):
     + LOOP_I_LOOP_J:
            * Trip count:     400
            * Latency:        401
            * Pipeline II:    1
            * Pipeline depth: 2
```

When the design contains nested loops, analyze the results to ensure as many nested loops as possible have been flattened: review the log file or look in the synthesis report for cases, as shown above, where the loop labels have been merged (`LOOP_I` and `LOOP_J` are now reported as `LOOP_I_LOOP_J`).

Loop Parallelism

Vivado HLS will schedule logic and functions are early as possible to reduce latency. To perform this it will schedule as many logic operations and functions as possible in parallel. It will not, however, schedule loops to execute in parallel.

If Example 2-44 is synthesized, loop `SUM_X` will be scheduled and then loop `SUM_Y` will be scheduled: even though loop `SUM_Y` does not need to wait for loop `SUM_X` to complete before it can begin its operation, it will be scheduled after `SUM_X`.

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
              dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     SUM_X:for (i=0;i<xlimit; i++) {
          X_accum += A[i];
          X[i] = X_accum;
}

     SUM_Y:for (i=0;i<ylimit; i++) {
          Y_accum += B[i];
          Y[i] = Y_accum;
     }
}
```

Example 4-44: **Sequential Loops**

Since the loops have different bounds (`xlimit` and `ylimit`) they cannot be merged. However by placing the loops in separate functions, as shown in Example 2-45, the exact same functionality can be achieved and both loops (inside the functions), can be scheduled in parallel.

```
#include "loop_functions.h"

void sub_func(din_t I[N], dout_t O[N], dsel_t limit) {
     int i;
     dout_t accum=0;

     SUM:for (i=0;i<limit; i++) {
            accum += I[i];
            O[i] = accum;
     }

}

void loop_functions(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
              dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     sub_func(A,X,xlimit);
     sub_func(B,Y,ylimit);
}
```

Example 4-45: **Sequential Loops as Functions**

If Example 2-45 is synthesized, the latency will be half the latency of Example 2-44 because the loops (as functions) can now execute in parallel.

The `dataflow` optimization could also be used in Example 2-44. The principle of capturing loops in functions to exploit parallelism is presented here for cases where `dataflow` optimization cannot be used. For example, in a larger example, `dataflow` optimization would be applied to all loops and functions at the top-level and memories placed between every top-level loop and function.

Loop Dependencies

Loop dependencies are data dependencies which prevent optimization of loops, typically pipelining. They can be within a single iteration of a loop and or between different iteration of a loop.

The simplest way to understand loop dependencies is to examine an extreme example. In the following example, the result of the loop is used as the loop continuation/exit condition: each iteration of the loop must finish before the next can start.

```
Minim_Loop: while (a != b) {
       if (a > b)
              a -= b;
       else
              b -= a;
}
```

In this case, this loop cannot be pipelined: the next iteration of the loop cannot begin until the previous iteration ends.

Not all loop dependencies are as extreme as this, but this example highlights the problem: some operation cannot begin until some other operation has completed. The solution is to try ensure the initial operation is performed as early as possible.

Loop dependencies can occur with any and all types of data; however they are particularly common when using arrays. As such, they are discussed in the next section on arrays.

Arrays

Arrays are typically implemented as a memory (RAM, ROM or FIFO) after synthesis. As discussed in the section Arrays on the Interface, arrays on the top-level function interface are synthesized as RTL ports which access a memory outside. Arrays internal to the design are synthesized to internal BRAM, LUTRAM or registers, depending on the optimization settings.

Like loops, arrays are an intuitive coding construct and so they are often found in C programs. Also like loops, Vivado HLS has a number of optimizations and directives which can be applied to optimize their implementation in RTL without any need to modify the code.

Cases where arrays can introduce problems in the RTL are:

- Array accesses can often create bottlenecks to performance. When implemented as a memory, the number of memory ports limits access to the data.
- Array initialization, if not performed carefully, can result in undesirably long reset and initialization in the RTL.
- Some care must be taken to ensure arrays which only require read accesses are implemented as ROMs in the RTL.

As discussed in Pointers, arrays of pointers are supported, however each pointer can only point to a scalar or an array of scalars.

Note: Arrays must be sized.
Supported: Array[10];
Not Supported: Array[];

Array Accesses

The code in Example 2-46 shows a case where accesses to an array can limit performance in the final RTL design. In this example there are three accesses to the array `mem[N]` to create a summed result.

```
#include "array_mem_bottleneck.h"

dout_t array_mem_bottleneck(din_t mem[N]) {

      dout_t sum=0;
      int i;

      SUM_LOOP:for(i=2;i<N;++i)
        sum += mem[i] + mem[i-1] + mem[i-2];

      return sum;
}
```

Example 4-46: **Array-Memory Bottleneck**

During synthesis the array is implemented as a RAM. If the RAM is specified as a single-port RAM it is impossible to pipeline loop `SUM_LOOP` to process a new loop iteration every clock cycle.

Trying to pipeline `SUM_LOOP` with an initiation interval of 1 will result in the following message (after failing to achieve a throughput of 1, Vivado HLS automatically relaxes the constraint):

```
@I [SCHED-61] Pipelining loop 'SUM_LOOP'.
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_1',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 1).
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_2',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 4.
```

The problem here is that the single-port RAM has only a single data port: only 1 read (and 1 write) can be performed in each clock cycle.

- SUM_LOOP Cycle1: read `mem[i]`;
- SUM_LOOP Cycle2: read `mem[i-1]`, sum values;
- SUM_LOOP Cycle3: read `mem[i-2]`, sum values;

A dual-port RAM could be used, but this will only allow two accesses per clock cycle. Three reads are required to calculate the value of sum and so three accesses per clock cycle are required in order to pipeline the loop with an new iteration every clock cycle.

CAUTION! *Arrays implemented as memory or memory ports, can often become bottlenecks to performance.*

The code in Example 2-46 can be re-written as shown in Example 2-47 to allow the code to be pipelined with a throughput of 1. Notice, in Example 2-47 by performing pre-reads and manually pipelining the data accesses there is only one array read specified in each iteration of the loop: this ensures only a single-port RAM is required to achieve the performance.

```
#include "array_mem_perform.h"

dout_t array_mem_perform(din_t mem[N]) {

      din_t tmp0, tmp1, tmp2;
      dout_t sum=0;
      int i;

      tmp0 = mem[0];
      tmp1 = mem[1];
      SUM_LOOP:for (i = 2; i < N; i++) {
             tmp2 = mem[i];
             sum += tmp2 + tmp1 + tmp0;
             tmp0 = tmp1;
             tmp1 = tmp2;
      }

      return sum;
}
```

Example 4-47: **Array-Memory with Performance Access**

Vivado HLS has a number of optimization directives for changing how arrays are implemented and accessed. It is typically the case that directives can be used, and changes to the code are not required. Arrays can be partitioned into blocks or into their individual elements. In some cases, Vivado HLS will automatically partition arrays into individual elements: this is controllable using the configuration settings for auto-partitioning.

When an array is partitioned into multiple blocks, the single array is implemented as multiple RTL RAM blocks. When partitioned into elements, each element will be implemented as a register in the RTL. In both cases, partitioning allows more elements to be accessed in parallel and can help with performance; the design trade-off is between performance and the number of RAMs or registers required to achieve it.

FIFO accesses

A special care of arrays accesses are when arrays are implemented as FIFOs. This is often the case when dataflow optimization is used.

Accesses to a FIFO must be in sequential order starting from location zero. In addition, if an array is read in multiple locations, the code must strictly enforce the order of the FIFO accesses. It is often the case that arrays with multiple fanout cannot be implemented as FIFOs without additional code to enforce the order of the accesses.

Array Initialization

RECOMMENDED: *As discussed in the Type Qualifiers section, although not a requirement, it is highly recommended to specify arrays which are to be implemented as memories with the static qualifier. This not only ensures Vivado HLS will implement the array with a memory in the RTL, it also allows the initialization behavior of static types to be used*

In the following code, an array is initialized with a set of values. Each time the function is executed, array `coeff` is assigned these values. After synthesis, each time the design executes the RAM which implements `coeff` will be loaded with these values. For a single-port RAM this would take 8 clock cycles. For an array of 1024, it would of course, take 1024 clock cycles, during which time no operations depending on `coeff` could occur.

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

The following code uses the `static` qualifier to define array `coeff`. The array is initialized with the specified values at start of execution. Each time the function is executed, however, array `coeff` remembers its values from the previous execution: a static array behaves in C code as a memory does in RTL.

```
static int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

In addition, if the variable has the `static` qualifier, Vivado HLS will initialize the variable in the RTL design and in the FPGA bitstream: this removes the need for multiple clock cycles to initialize the memory and ensures that initializing large memories is not an operational overhead.

The RTL configuration command can be used to specify if static variables return to their initial state after a reset is applied (not the default). If a memory is to be returned to its initial state after a reset operation, this will incur an operational overhead and require multiple cycles to reset the values: each value has to be written into each memory address.

Implementing ROMs

As was shown in Example 2-29 in the review of `static` and `const` type qualifiers, Vivado HLS does not require than an array be specified with the `static` qualifier in order to synthesize a memory or the `const` qualifier in order to infer the memory should be a ROM. Vivado HLS will perform analysis of the design and seek to create the most optimum hardware.

It is however highly recommended to use the static qualifier for arrays which are intended to be memories: as noted in Array Initialization, a static type behaves in an almost identical manner as a memory in RTL.

The `const` qualifier is also recommended when arrays are only read, since Vivado HLS cannot always infer a ROM should be used by analysis of the design. The general rule for

the automatic inference of a ROM is that a local, static (non-global) array is written to before being read. The following practices in the code can help infer a ROM:

- Initialize the array as early as possible in the function that uses it.
- Group writes together.
- Do not interleave `array(ROM)` initialization writes with non-initialization code.
- Do not store different values to the same array element (group all writes together in the code).
- Element value computation must not depend on any non-constant (at compile-time) design variable(s), other that the initialization loop counter variable.

If complex assignments are used to initialize a ROM, for example functions from the `math.h` library, placing the array initialization into a separate function will allow a ROM to be inferred. In Example 2-48, array `sin_table[256]` is inferred as a memory and implemented as a ROM after RTL synthesis.

```
#include "array_ROM_math_init.h"
#include <math.h>

void init_sin_table(din1_t sin_table[256])
{
      int i;
      for (i = 0; i < 256; i++) {
            dint_t real_val = sin(M_PI * (dint_t)(i - 128) / 256.0);
            sin_table[i] = (din1_t)(32768.0 * real_val);
      }
}

dout_t array_ROM_math_init(din1_t inval, din2_t idx)
{
      short sin_table[256];
      init_sin_table(sin_table);
      return (int)inval * (int)sin_table[idx];
}
```

Example 4-48: **ROM Initialization with math.h**

TIP: *Since the result of the `sin()` function results in constant values, no core is required in the RTL design to implement the `sin()` function. The `sin()` function is not one of the cores listed in Table 2-2 and is not supported for synthesis in C. (Refer to the C++ for Synthesis section for using `math.h` functions in C++.)*

Unsupported C Constructs

While Vivado HLS has support for a wide range of the C language, there are some constructs which are not synthesizable or result in errors further down the design flow. This section outlines areas where coding changes must be made, if the function is to be synthesized and implemented in an FPGA device.

As a general rule, in order to be synthesized the C function must contain the entire functionality of the design (none of the functionality can be performed by system calls to the operating system), the C constructs must be of a fixed/bounded size and the implementation of those constructs unambiguous.

System Calls

System calls cannot be synthesized since they are actions which relate to performing some task upon the operating system in which the C program is running.

Vivado HLS will automatically ignore commonly used system calls which only display data and have no impact on the execution of the algorithm, such as `printf()` and `fprintf(stdout,)`, however in general calls to the system cannot be synthesized and should be removed from the function prior to synthesis. Other examples of such calls are `getc()`, `time()`, `sleep()`, etc. all of which make calls to the operating system.

Vivado HLS automatically defines the `macro __SYNTHESIS__` when synthesis is performed. This allows the `__SYNTHESIS__ macro` to be used to exclude non-synthesizable code from the design.

Example 2-49 shows a case where the intermediate results from a sub-function are saved to a file on the hard drive. The macro `__SYNTHESIS__` is used to ensure the non-synthesizable files writes are ignored during synthesis.

```
#include "hier_func4.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
     *outSum = *in1 + *in2;
     *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
     *outA = *in1 >> 1;
     *outB = *in2 >> 2;
}

void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
     dint_t apb, amb;

     sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
     FILE *fp1;// The following code is ignored for synthesis
     char filename[255];
     sprintf(filename,"Out_apb_%03d.dat",apb);
     fp1=fopen(filename,"w");
     fprintf(fp1, "%d \n", apb);
     fclose(fp1);
#endif
     shift_func(&apb,&amb,C,D);
}
```

Example 4-49: **File Writes for Debug**

The `__SYNTHESIS__` macro is provided as a convenient way to exclude non-synthesizable code, without removing the code itself from the C function. Using such a macro does however mean the C code for simulation and the C code for synthesis are now different.

CAUTION! *If the `__SYNTHESIS__` macro is used to change the functionality of the C code, it can result in different results between C simulation and C synthesis. Errors in such code are inherently difficult to debug, and using the `__SYNTHESIS__` macro to create changes in functionality should be avoided.*

Dynamic Memory Usage

Any system calls which manage memory allocation within the system, for example, `malloc()`, `alloc()`, and `free()` are using resources which exist in the memory of the operating system and are created and released during runtime: to be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

Memory allocation system calls must be removed from the design code prior to synthesis. However, since dynamic memory operations are used to define the functionality of the design, they must be transformed into equivalent bounded representations. Example 2-50 shows how a design using `malloc()` can be transformed into a synthesizable version.

The code in Example 2-50 highlights two useful coding style techniques:

- First, the design does not make use of the `__SYNTHESIS__` macro: rather the user defined macro `NO_SYNTH` is used to select between the synthesizable and non-synthesizable versions. This ensures the same exact same code is simulated in C and synthesized in Vivado HLS.
- Secondly, the pointers in the original design using `malloc()` do not need to be re-written to work with fixed sized elements. Fixed sized resources can be created and the existing pointer can simply be made to point to the fixed sized resource: this technique can prevent manual re-coding of the existing design.

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
     long long *out_accum = malloc (sizeof(long long));
     int* array_local = malloc (64 * sizeof(int));
#else
     long long _out_accum;
     long long *out_accum = &_out_accum;
     int _array_local[64];
     int* array_local = &_array_local[0];
```

```
#endif
      int i,j;

      LOOP_SHIFT:for (i=0;i<N-1; i++) {
            if (i<width)
                  *(array_local+i)=din[i];
            else
                  *(array_local+i)=din[i]>>2;
      }

      *out_accum=0;
      LOOP_ACCUM:for (j=0;j<N-1; j++) {
            *out_accum += *(array_local+j);
      }

      return *out_accum;
}
```

Example 4-50: **Transforming malloc() to Fixed Resources**

RECOMMENDED: *Since the coding changes here impact the functionality of the design, it is not recommended to use the `__SYNTHESIS__` macro. The recommended approach is to:*

1. Add the user defined macro `NO_SYNTH` to the code and modify the code.
2. Enable macro `NO_SYNTH`, execute the C simulation and save the results.
3. Disable the macro `NO_SYNTH` (e.g. comment out, as in Example 50), execute the C simulation to verify the results are identical.

Perform synthesis with the user defined macro disabled.
This methodology ensures the updated code is validated with C simulation and the exact same code is then synthesized.

Pointer Limitations

General Pointer Casting

Pointer casting is not supported in the general case but is supported between native C types. Refer to the Pointers section for details on pointer casting.

Pointer Arrays

Arrays of pointers are supported for synthesis if each pointer points to a scalar or an array of scalars. Arrays of pointers cannot point to additional pointers. Refer to the Pointers section for details on pointer arrays.

Recursive Functions

Recursive functions cannot be synthesized. This applies to functions which can form endless recursion, where endless :

```
unsigned foo (unsigned n)
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Tail recursion, where there are a finite number of function calls, is also not supported:

```
unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
    if (n == 0) return m;
    return foo(n, m%n);
}
```

In C++, templates can be used to implement tail recursion. C++ is addressed next.

C++ for Synthesis

This chapter covers aspects of the C++ language as it is used for synthesis using Vivado HLS. Almost all of the items covered in the chapter on C for Synthesis also relate to coding with C++ (top-level function arguments, pointers, loops, arrays etc.) and the chapter C for Synthesis should have be read before proceeding with this chapter.

Topics in C for Synthesis which do not apply when using C++ are the arbitrary precision types used with C (C++ has its own arbitrary precision types) and the limitations when compiling arbitrary precision types in C: `apcc` is not required for C++ simulation and no Integer Promotion Issues are encountered with C++ arbitrary precision types.

The addition language features of C++, relevant for synthesis, include classes, templates, C++ arbitrary precision types, support for the `math.h` library and standard template libraries are covered in this chapter.

Vivado HLS expects C++ functions to be named with the standard `g++` file extensions (`.cpp`, `.cxx`, etc). Standard C language functions, appropriately renamed and not using Vivado HLS C arbitrary precision types `(u)int#`, can be synthesized as C++ designs. There is no requirement to use a C++ object orientated coding style.

C++ Classes

C++ classes are fully supported for synthesis with Vivado HLS. The top-level for synthesis must be a function: a class cannot be the top-level for synthesis. To synthesize a class member function, the class itself should be instantiated into function: the top-level class should not simply be instantiated into the test bench. Example 2-51 shows how class `CFIR` (defined in the header file discussed next) is instantiated in the top-level function `cpp_FIR` and used to implement an `FIR` filter.

```
#include "cpp_FIR.h"
```

```
// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
     {
       static CFir<coef_t, data_t, acc_t> fir1;

       cout << fir1;

       return fir1(x);
     }
```

Example 4-51: **C++ FIR Filter**

IMPORTANT: *Classes and class member functions cannot be the top-level for synthesis. The class should be instantiated in a top-level function.*

Before examining the class used to implement the design in Example 2-51, it is worth noting Vivado HLS automatically ignores the standard output stream cout during synthesis. When synthesized Vivado HLS will issue the following warnings:

```
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call: 'std::operator<<
<std::char_traits<char> >' (cpp_FIR.h:110)
@
```

The header file `cpp_FIR.h` is shown below in Example 2-52 and shows the definition of class `CFir` and its associated member functions. In this example the operator member functions `()` and `<<` are overloaded operators, which are respectively used to execute the main algorithm and used with `cout` to format the data for display during C simulation.

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define N 85

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

// Class CFir definition
template<class coef_T, class data_T, class acc_T>
class CFir {
     protected:
            static const coef_T c[N];
            data_T shift_reg[N-1];
     private:
     public:
            data_T operator()(data_T x);
            template<class coef_TT, class data_TT, class acc_TT>
```

```
            friend ostream&
            operator<<(ostream& o, const CFir<coef_TT, data_TT, acc_TT> &f);
};

// Load FIR coefficients
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[N] = {
     #include "cpp_FIR.inc"
};

// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
data_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {
     int i;
     acc_t acc = 0;
     data_t m;

     loop: for (i = N-1; i >= 0; i--) {
            if (i == 0) {
                  m = x;
                  shift_reg[0] = x;
            } else {
                  m = shift_reg[i-1];
                  if (i != (N-1))
                  shift_reg[i] = shift_reg[i - 1];
            }
            acc += m * c[i];
     }
     return acc;
}

// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T, data_T, acc_T> &f) {
     for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i++) {
            o << "shift_reg[" << i << "]= " << f.shift_reg[i] << endl;
     }
     o << "________________" << endl;
     return o;
}

data_t cpp_FIR(data_t x);
```

Example 4-52: **C++ Header File Defining Classes**

The test bench Example 2-51 is shown in Example 2-53 and demonstrates how top-level function `cpp_FIR` is called and validated. This example highlights some of the important attributes of a good test bench for Vivado HLS synthesis:

- The output results are checked against known good values.
- The test bench returns 0 if the results are confirmed to be correct.

More details on test benches are provided in the Creating of Productive Test Bench section.

```
#include "cpp_FIR.h"
```

```
int main() {
      ofstream result;
      data_t output;
      int retval=0;


      // Open a file to save the results
      result.open("result.dat");

      // Apply stimuli, call the top-level function and save the results
      for (int i = 0; i <= 250; i++)
      {
            output = cpp_FIR(i);

            result << setw(10) << i;
            result << setw(20) << output;
            result << endl;

      }
      result.close();

      // Compare the results file with the golden results
      retval = system("diff --brief -w result.dat result.golden.dat");
      if (retval != 0) {
            printf("Test failed  !!!\n");
            retval=1;
      } else {
            printf("Test passed !\n");
      }

      // Return 0 if the test
      return retval;
}
```

Example 4-53: **C++ Test Bench for cpp_FIR**

Constructors, Destructors and Virtual Functions

Class constructors and destructors will be included and synthesized whenever a class object is declared.

Virtual functions, including abstract ones, are supported for synthesis if Vivado HLS can statically determine the function during elaboration. The following are cases where virtual functions are not supported for synthesis:

- Virtual functions can be defined in a multi-layer inheritance class hierarchy but only with a single inheritance.
- Dynamic polymorphism is only supported if the pointer object can be determined at compile time. For example, such pointers cannot be used in an if-else or loop constructs.
- An STL container cannot be used to contain the pointer of an object and call the polymorphism function. For example:

```
vector<base *> base_ptrs(10);

//Push_back some base ptrs to vector.
for (int i = 0; i < base_ptrs.size(); ++i) {
     //Static elaboration cannot resolve base_ptrs[i] to actual data type.
     base_ptrs[i]->virtual_function();
}
```

- Cases where the base object pointer is a global variable are not supported. For example:

```
Base *base_ptr;

void func()
{
     ……
     base_prt->virtual_function();
     ……
}
```

- The base object pointer cannot be a member variable in a class definition.

```
// Static elaboration cannot bind base object pointer with correct data type.
class A
{
     …..
     Base *base_ptr;
     void set_base(Base *base_ptr);
     void some_func();
     …..
};

void A::set_base(Base *ptr)
{
     this.base_ptr = ptr;
}

void A::some_func()
{
     ….
     base_ptr->virtual_function();
     ….
}
```

- If the base object pointer or reference is in the function parameter list of constructor, Vivado HLS will not convert it. (The ISO C++ standard has depicted this in section12.7: sometimes the behavior is undefined).

```
class A {
     A(Base *b) {
            b-> virtual _ function ();
     }
};
```

Global Variables and Classes

It is not recommended to use global variables in classes as they can prevent some optimizations from occurring. Example 2-54, shows a case where a class is used to create

the component for a filter (class `polyd_cell` is used as a component which performs shift, multiply and accumulate operations).

```
typedef long long   acc_t;
typedef int  mult_t;
typedef char data_t;
typedef char coef_t;

#define TAPS          3
#define PHASES        4
#define DATA_SAMPLES 256
#define CELL_SAMPLES 12

// Use k on line 73 static int k;

template <typename T0, typename T1, typename T2, typename T3, int N>
class polyd_cell {
private:
public:
     T0 areg;
     T0 breg;
     T2 mreg;
     T1 preg;
  T0 shift[N];
     int k;   //line 73
  T0 shift_output;
     void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
     {
     Function_label0:;

     if (col==0) {
            SHIFT:for (k = N-1; k >= 0; --k) {
                   if (k > 0)
                          shift[k] = shift[k-1];
                   else
                          shift[k] = data;
            }
     *dataOut = shift_output;
     shift_output = shift[N-1];
     }
     *pcout = (shift[4*col]* coeff) + pcin;

     }
};

// Top-level function with class instantiated
void cpp_class_data (
     acc_t          *dataOut,
     coef_t         coeff1[PHASES][TAPS],
     coef_t         coeff2[PHASES][TAPS],
     data_t  dataIn[DATA_SAMPLES],
     int            row
) {

     acc_t  pcin0 = 0;
     acc_t  pcout0, pcout1;
     data_t dout0,  dout1;
     int col;
```

```
    static acc_t   accum=0;
    static int sample_count = 0;
    static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell0;
    static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell1;

    COL:for (col = 0; col <= TAPS-1; ++col) {

    polyd_cell0.exec(&pcout0,&dout0,pcin0,coeff1[row][col],dataIn[sample_count],
col);

    polyd_cell1.exec(&pcout1,&dout1,pcout0,coeff2[row][col],dout0,col);


          if ((row==0) && (col==2)) {
                *dataOut = accum;
                accum = pcout1;
          } else {
                accum = pcout1 + accum;
          }

    }
    sample_count++;
}
```

Example 4-54: **C++ Class Data Member used for Loop Index**

Within class `polyd_cell` there is a loop `SHIFT` used to shift data. If the loop index `k` used in loop `SHIFT` was removed and replaced with the global index for `k` (shown earlier in the example, but commented `static int k`), Vivado HLS would be unable to pipeline any loop or function in which class `polyd_cell` was used. Vivado HLS would issue the following message:

```
@W [XFORM-503] Cannot unroll loop 'SHIFT' in function 'polyd_cell<char, long long,
int, char, 12>::exec' completely: variable loop bound.
```

Using local non-global variables for loop indexing ensures Vivado HLS can perform all optimizations.

Templates

As earlier examples in this chapter have shown, Vivado HLS supports the use of templates in C++ for synthesis. Templates are not supported however for the top-level function.

IMPORTANT: *The top-level function cannot be a template.*

In addition to the general use of templates shown in Example 2-52 and Example 2-54, templates can be used implement a form of recursion, which is not supported in standard C synthesis (Recursive Functions).

Example 2-55 shows a case where a templatized `struct` is used to implement a tail-recursion Fibonacci algorithm. The key to performing synthesis is that a termination class is used to implement the final call in the recursion, where a template size of one is used.

```
//Tail recursive call
template<data_t N> struct fibon_s {
     template<typename T>
     static T fibon_f(T a, T b) {
            return fibon_s<N-1>::fibon_f(b, (a+b));
     }
};

// Termination condition
template<> struct fibon_s<1> {
     template<typename T>
     static T fibon_f(T a, T b) {
            return b;
     }
};

void cpp_template(data_t a, data_t b, data_t &dout){
     dout = fibon_s<FIB_N>::fibon_f(a,b);
}
```

Example 4-55: **C++ Tail Recursion with Templates**

Types

As with the C language types discussed in section Types, Vivado HLS supports the same standard types in C++ types for synthesis.

Support is also provided for C++ arbitrary precision integers: the C++ arbitrary precision integers are not the same as those used in C and do not have any of the simulation limitations. In addition supported is provided in C++ for arbitrary precision fixed point types.

C++ Arbitrary Precision Integer Types

The native data types in C++ are on 8-bit boundaries (8, 16, 32 and 64 bits). RTL signals and operations however support arbitrary bit-lengths.

Vivado HLS provides arbitrary precision data types for C++ to allow variables and operations in the C++ code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit etc. up to 1024 bits.

TIP: *The default maximum width allowed is 1024 bits.This default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 32768 before inclusion of the ap_int.h header file.*

C++ supports use of the arbitrary precision types defined in the SystemC standard: simply include the SystemC header file systemc.h and use SystemC data types. More details on SystemC types are provided in the chapter on SystemC.

Arbitrary precision data types have are two primary advantages over the native C++ types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can be used to specify that exactly 17-bit are used in the calculation.
    - Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP48 components.
- Accurate C++ simulation/analysis: Arbitrary precision data types in the C++ code allows the C++ simulation to be performed using accurate bit-widths and for the C++ simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The arbitrary precision types in C++ have none of the disadvantages of those in C:

- C++ arbitrary types can be compiled with standard C++ compilers (there is no C++ equivalent of apcc, as discussed in Validating Arbitrary Precision Types in C).
- C++ arbitrary precision types do not suffer from Integer Promotion Issues.

It is not uncommon for users to a file extension from `.c` to `.cpp` so the file can be compiled as C++, where neither of the above issues are present.

The remainder of this section explains how to use arbitrary precision types. A detailed description of arbitrary precision types is provided in a reference section at the end of this document (C++ Arbitrary Precision Types) and includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A description of Vivado HLS helper methods, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift values, results in a shift in the opposite direction).

Using Arbitrary Precision Types with C++

For the C++ language, the header file `ap_int.h` defines the arbitrary precision integer data types `ap_(u)int<W>`. For example, `ap_int<8>` represents an 8-bit signed integer data type and `ap_uint<234>` represents a 234-bit unsigned integer type.

The ap_int.h file is located in the directory $HLS_ROOT/include, where $HLS_ROOT is the HLS installation directory.

The code shown in Example 2-56, is a repeat of the code shown in the earlier example on basic arithmetic (Example 2-22 and again in Example 2-35). In this example the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t`, etc.

```
#include "cpp_ap_int_arith.h"

void cpp_ap_int_arith(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
            dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
) {

     // Basic arithmetic operations
     *out1 = inA * inB;
     *out2 = inB + inA;
     *out3 = inC / inA;
     *out4 = inD % inA;

}
```

Example 4-56: **Basic Arithmetic Revisited with C++ Types**

In this latest update to this example, the C++ arbitrary precision types are used:

- Add header file ap_int.h to the source code.
- Change the native C++ types to arbitrary precision types `ap_int<N>` or `ap_uint<N>`, where `N` is a bit-size from 1 to 1024 (as noted above, this can be extended to 32K-bits is required).

The data types are defined in the header `cpp_ap_int_arith.h` as shown in Example 2-36.

Compared with Example 2-22, the input data types have simply been reduced to represent the maximum size of the real input data (e.g. 8-bit input `inA` is reduced to 6-bit input). The output types however have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_

#include <stdio.h>
#include "ap_int.h"

#define N 9

// Old data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef ap_int<6> dinA_t;
```

```
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;

typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;

void cpp_ap_int_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);

#endif
```

Example 4-57: **Basic Arithmetic with C++ Arbitrary Precision Types**

If Example 2-56 is synthesized it will result in a design which is functionally identical to Example 2-22 and Example 2-36: to keep the test bench as similar as possible to Example 2-36, rather than use the C++ cout operator to output the results to a file, the built-in `ap_int` method `.to_int()` is used to convert the `ap_int` results to integer types used with the standard `fprintf` function.

```
fprintf(fp, "%d*%d=%d; %d+%d=%d; %d/%d=%d; %d mod %d=%d;\n",
     inA.to_int(), inB.to_int(), out1.to_int(),
     inB.to_int(), inA.to_int(), out2.to_int(),
     inC.to_int(), inA.to_int(), out3.to_int(),
     inD.to_int(), inA.to_int(), out4.to_int());
```

Note: Section C++ Arbitrary Precision Types provides comprehensive details on the methods, synthesis behavior and all aspects of using the `ap_(u)int<N>` arbitrary precision data types.

C ++Arbitrary Precision Fixed Point Types

C++ functions can take advantage of the arbitrary precision fixed point types provided with Vivado HLS. Figure 2-9 summarizes the basic features of these fixed point types:

- The word can be signed (`ap_fixed`) or unsigned (`ap_ufixed`).
- A word with of any arbitrary size `W` can be defined.
- The number of places above the decimal point `I`, also defines the number of decimal places in the word, `W-I` (represented by `B` in Figure 2-9).
- The type of rounding or quantization (`Q`) can be selected.
- The overflow behavior (`O` and `N`) can be selected.

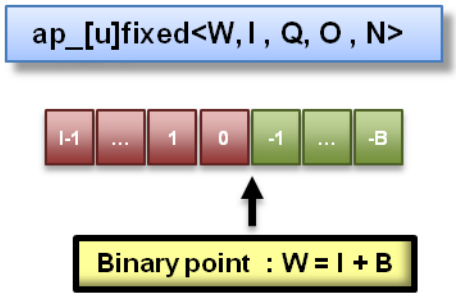

Figure 4-9: **Arbitrary Precision Fixed Point Types**

The arbitrary precision fixed point types can be used when header file `ap_fixed.h` is included in the code.

The advantages of using fixed point types are:

- They allow fractional number to be easily represented.
- When variables have a different number of integer and decimal place bits, the alignment of the decimal point is handled automatically.
- There are numerous options to automatically handle how rounding should happen: when there are too few decimal bits to represent the precision of the result.
- There are numerous options to automatically handle how variables should overflow: when the result is greater than the number of integer bits can represent.

These attributes are summarized by examining the code in Example 2-58. First, the header file `ap_fixed.h` is included. The `ap_fixed` types are then defined via `typedef` statement:

- A 10-bit input: 8-bit integer value with 2 decimal places.
- A 6-bit input: 3-bit integer value with 3 decimal places.
- A 22-bit variable for the accumulation: 17-bit integer value with 5 decimal places.
- A 36-bit variable for the result: 30-bit integer value with 6 decimal places.

Notice the function contains no code to manage the alignment of the decimal point after operations are performed: that is done automatically.

```
#include "ap_fixed.h"

typedef ap_ufixed<10,8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6,3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2) {

     static dint_t sum;
     sum =+ d_in1;
```

```
        return sum * d_in2;
}
```

Example 4-58: **AP_Fixed Point Example**

The quantization and overflow modes are shown in Table 2-10 and are described in detail in the in the reference section C++ Arbitrary Precision Fixed Point Types.

TIP: *Quantization and overflow modes which do more than the default behavior of standard hardware arithmetic (wrap and truncate) will result in operators with more associated hardware: it costs logic (LUTs) to implement the more advanced modes, such as round to minus infinity or saturate symmetrically.*

Table 4-10: **Fixed Point Identifier Summary**

Identifier	Description	
W	Word length in bits	
I	The number of bits used to represent the integer value (the number of bits above the decimal point)	
Q	Quantization mode dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	Mode	**Description**
	AP_RND	Rounding to plus infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode dictates the behavior when more bits are generated than the variable to store the result contains.	
	Mode	**Description**
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.	

Using `ap_(u)fixed` types the C++ simulation will be bit-accurate and fast simulation can be used to validate the algorithm and its accuracy. After synthesis, the RTL will exhibit the exact same bit-accurate behavior.

Arbitrary precision fixed point types can be freely assigned literal values in the code, as shown in the test bench (Example 2-59) used with Example 2-58, where the values of `in1` and `in2` are declared and assigned constant values.

When assigning literal values involving operators the literal values must first be cast to `ap_(u)fixed` types or the C compiler and Vivado HLS will interpret the literal as an integer or `float/double` type and may fail to find a suitable operator. For example, in the assignment of `in1 = in1 + din1_t(0.25)` the literal 0.25 is cast an `ap_fixed` type.

```
int main()
  {
      ofstream result;
      din1_t in1 = 0.25;
      din2_t in2 = 2.125;
      dout_t output;
      int retval=0;


      result.open("result.dat");
      // Persistent manipulators
      result << right << fixed << setbase(10) << setprecision(15);

      for (int i = 0; i <= 250; i++)
      {
             output = cpp_ap_fixed(in1,in2);

             result << setw(10) << i;
             result << setw(20) << in1;
             result << setw(20) << in2;
             result << setw(20) << output;
             result << endl;

             in1 = in1 + din1_t(0.25);
             in2 = in2 - din2_t(0.125);
      }
      result.close();

      // Compare the results file with the golden results
      retval = system("diff --brief -w result.dat result.golden.dat");
      if (retval != 0) {
             printf("Test failed  !!!\n");
             retval=1;
      } else {
             printf("Test passed !\n");
      }

      // Return 0 if the test passes
      return retval;
}
```

Example 4-59: **AP_Fixed Point Test Bench Example**

Unsupported C++ Constructs

The supported C++ constructs which cannot be synthesized are listed in this section and are in addition to those listed in Unsupported C Constructs.

Dynamic Objects

As with restrictions on dynamic memory usage in C, C++ objects which are dynamically created and/or destroyed are not supported for synthesis. This includes dynamic polymorphism and dynamic virtual function calls. The following cannot be synthesized since it create new function at run time.

```
Class A {
public:
     virtual void bar() {…};
};

void fun(A* a) {
     a->bar();
}
A* a = 0;
if (base)
     A= new A();
else
     A = new B();

foo(a);
```

Standard Template Libraries

Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation. For this reason the STLs cannot be synthesized. The solution with STLs is to create a local function with identical functionality which does not exhibit these characteristics of recursion, dynamic memory allocation or the dynamic creation and destruction of objects.

SystemC Synthesis

Vivado HLS provides support for SystemC (IEEE standard 1666), a C++ class library used to model hardware and available at www.systemc.org. Vivado HLS supports SystemC version 2.1 and SystemC Synthesizable Subset (Draft 1.3).

This section provides details on the synthesis of SystemC functions with Vivado HLS. The information provided here is in addition to the information provided in the earlier chapters, C for Synthesis and C++ for Synthesis, and those chapters should be read to fully understand the basic rules of coding for synthesis.

IMPORTANT: *As with C and C++ designs, the top-level function for synthesis must be a function below the top-level for C compilation* `sc_main()`*: the* `sc_main()` *function cannot be the top-level function for synthesis.*

Design Modeling

The top-level for synthesis must be an `SC_MODULE`. Designs can be synthesized if modeled using the SystemC constructor processes `SC_METHOD` and `SC_CTHREAD` or if `SC_MODULES` are instantiated inside other `SC_MODULES`.

An SC_ `MODULE` cannot be defined inside another `SC_MODULE` (they can be instantiated, as shown later). In cases, like the following where a module is defined inside another:

```
SC_MODULE(nested1)
{
      SC_MODULE(nested2)
      {
            sc_in<int> in0;
            sc_out<int> out0;
            SC_CTOR(nested2)
            {
                  SC_METHOD(process);
                  sensitive<<in0;
            }
            void process()
            {
                  int var =10;
                  out0.write(in0.read()+var);
            }
      };

      sc_in<int> in0;
      sc_out<int> out0;
      nested2 nd;
      SC_CTOR(nested1)
      :nd("nested2")
      {
            nd.in0(in0);
            nd.out0(out0);
      }
};
```

Must be transformed into a version, such as shown next, where the modules are not nested.

```
SC_MODULE(nested2)
{
      sc_in<int> in0;
      sc_out<int> out0;
      SC_CTOR(nested2)
      {
            SC_METHOD(process);
            sensitive<<in0;
      }
      void process()
```

```
        {
                int var =10;
                out0.write(in0.read()+var);
        }
};

SC_MODULE(nested1)
{
      sc_in<int> in0;
      sc_out<int> out0;
      nested2 nd;
      SC_CTOR(nested1)
      :nd("nested2")
      {
              nd.in0(in0);
      nd.out0(out0);
      }
};
```

Similarly, an SC_MODULE cannot be derived from another SC_MODULE, as shown in this example:

```
SC_MODULE(BASE)
{
      sc_in<bool> clock; //clock input
      sc_in<bool> reset;
      SC_CTOR(BASE) {}

};

class DUT: public BASE
{
public:
      sc_in<bool> start;
      sc_in<sc_uint<8> > din;
      …
};
```

RECOMMENDED: *Define the module constructor inside the module.*

Cases like the following,

```
SC_MODULE(dut) {
      sc_in<int> in0;
      sc_out<int>out0;
      SC_HAS_PROCESS(dut);
      dut(sc_module_name nm);
       …
};

dut::dut(sc_module_name nm)
{
      SC_METHOD(process);
      sensitive<<in0;
}
```

Should be transformed to:

```
SC_MODULE(dut) {
      sc_in<int> in0;
      sc_out<int>out0;

      SC_HAS_PROCESS(dut);
      dut(sc_module_name nm)
      :sc_module(nm)
      {
            SC_METHOD(process);
            sensitive<<in0;
      }
      …
};
```

SC_THREADs are not supported for synthesis.

Using SC_METHOD

Example 2-60 shows the header file, `sc_combo_method.h`, for a small combinational design modeled using an `SC_METHOD` to model a half-adder. The top-level design name, `sc_combo_method`, is specified in the `SC_MODULE`.

```
#include <systemc.h>

SC_MODULE(sc_combo_method){
      //Ports
      sc_in<sc_uint<1> > a,b;
      sc_out<sc_uint<1> > sum,carry;

      //Process Declaration
      void half_adder();

      //Constructor
      SC_CTOR(sc_combo_method){

            //Process Registration
            SC_METHOD(half_adder);
            sensitive<<a<<b;
      }
};
```

Example 4-60: **SystemC Combinational Example Header**

The design has two single-bit input ports (`a` and `b`). The `SC_METHOD` is sensitive to any changes in the state of either input port and executes function half_adder. The function `half_adder` is specified in file, `sc_combo_method.cpp`, shown in Example 2-61 and calculates the value for output port carry.

```
#include "sc_combo_method.h"

void sc_combo_method::half_adder(){
  bool s,c;
  s=a.read() ^ b.read();
```

```
  c=a.read() & b.read();
  sum.write(s);
  carry.write(c);

#ifndef __SYNTHESIS__
     cout << "Sum is " << a << " ^ " << b << " = " << s << ": " <<
sc_time_stamp() <<endl;
     cout << "Car is " << a << " & " << b << " = " << c << ": " <<
sc_time_stamp() <<endl;
#endif
```

Example 4-61: **SystemC Combinational Example Main Function**

Example 2-61 shows how any `cout` statements used to display values during C simulation can be protected from synthesis using the `__SYNTHESIS__` macro.

The test bench for the Example 2-61 is shown in Example 2-62. This test bench displays a number of important attributes required when using Vivado HLS.

```
#ifdef __RTL_SIMULATION__
#include "sc_combo_method_rtl_wrap.h"
#define sc_combo_method sc_combo_method_RTL_transactor
#else
#include "sc_combo_method.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"

int sc_main (int argc , char *argv[])
{
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);

     sc_signal<bool>     s_reset;
     sc_signal<sc_uint<1> >     s_a;
     sc_signal<sc_uint<1> >     s_b;
     sc_signal<sc_uint<1> >     s_sum;
     sc_signal<sc_uint<1> >     s_carry;

     // Create a 10ns period clock signal
     sc_clock s_clk("s_clk",10,SC_NS);

     tb_init       U_tb_init("U_tb_init");
     sc_combo_method     U_dut("U_dut");
     tb_driver     U_tb_driver("U_tb_driver");

     // Generate a clock and reset to drive the sim
     U_tb_init.clk(s_clk);
     U_tb_init.reset(s_reset);

     // Connect the DUT
     U_dut.a(s_a);
     U_dut.b(s_b);
     U_dut.sum(s_sum);
     U_dut.carry(s_carry);
```

```
        // Drive stimuli from dat* ports
        // Capture results at out* ports
        U_tb_driver.clk(s_clk);
        U_tb_driver.reset(s_reset);
        U_tb_driver.dat_a(s_a);
        U_tb_driver.dat_b(s_b);
        U_tb_driver.out_sum(s_sum);
        U_tb_driver.out_carry(s_carry);

        // Sim for 200
        int end_time = 200;

        cout << "INFO: Simulating " << endl;

        // start simulation
        sc_start(end_time, SC_NS);

        if (U_tb_driver.retval != 0) {
               printf("Test failed  !!!\n");
        } else {
               printf("Test passed !\n");
        }
        return U_tb_driver.retval;
};
```

Example 4-62: **SystemC Combinational Example Test Bench**

In order to perform RTL simulation using the `cosim_design` feature in Vivado HLS, the test bench must contain the macros shown at the top of Example 2-62. Given a design with the name `DUT`, the following must be used, where `DUT` is replaced with the actual design name.

```
#ifdef __RTL_SIMULATION__
#include "DUT_rtl_wrap.h"
#define DUT DUT_RTL_transactor
#else
#include "DUT.h"  //Original unmodified code
#endif
```

Failure to add this in the test bench where the design header file is included will result in `cosim_design` RTL simulation failing.

The report handler functions shown in Example 2-62 should be added to all SystemC test bench files used with Vivado HLS.

```
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);
```

These settings prevent the printing of extraneous messages during RTL simulation.

The most important of these messages are the warnings:

```
Warning: (W212) sc_logic value 'X' cannot be converted to bool
```

The adapters placed around the synthesized design will start with unknown (X) values. Not all SystemC types support unknown (X) values. This warning is issued when this occurs but if can be ignored unless the design lacks the appropriate data handshakes (and reads data unknown data).

Finally, the test bench in Example 2-62 performs checking on the results and returns a value of zero if the results are correct. In this case, the results are verified inside function `tb_driver` but the return value is checked and returned in the top-level test bench.

```
if (U_tb_driver.retval != 0) {
            printf("Test failed  !!!\n");
     } else {
            printf("Test passed !\n");
  }
  return U_tb_driver.retval;
```

Instantiating SC_MODULES

Hierarchical instantiations of `SC_MODULE`s can be synthesized, as shown in Example 2-63. In Example 2-63, the two instances of the half-adder design (`sc_combo_method`) from Example 2-60 are instantiated to create a full-adder design.

```
#include <systemc.h>
#include "sc_combo_method.h"

SC_MODULE(sc_hier_inst){
     //Ports
     sc_in<sc_uint<1> > a, b, carry_in;
     sc_out<sc_uint<1> > sum, carry_out;

     //Variables
     sc_signal<sc_uint<1> > carry1, sum_int, carry2;

     //Process Declaration
     void full_adder();

     //Half-Adder Instances
     sc_combo_methodU_1, U_2;

     //Constructor
     SC_CTOR(sc_hier_inst)
     :U_1("U_1")
     ,U_2("U_2")
     {
            // Half-adder inst 1
            U_1.a(a);
            U_1.b(b);
            U_1.sum(sum_int);
            U_1.carry(carry1);

            // Half-adder inst 2
            U_2.a(sum_int);
            U_2.b(carry_in);
```

```
            U_2.sum(sum);
            U_2.carry(carry2);

            //Process Registration
            SC_METHOD(full_adder);
            sensitive<<carry1<<carry2;
      }
};
```

Example 4-63: **SystemC Hierarchical Example**

The function `full_adder` is used to create the logic for the `carry_out` signal, as shown in Example 2-64.

```
#include "sc_hier_inst.h"

void sc_hier_inst::full_adder(){
  carry_out= carry1.read() | carry2.read();

}
```

Example 4-64: **SystemC full_adder Function**

Using SC_CTHREAD

The constructor process `SC_CTHREAD` is used to model clocked processes (threads) and is the primary way to model sequential designs. Example 2-65 shows a case which highlights the primary attributes of a sequential design.

- The data has associated handshake signals, allowing it to operate with the same test bench before and after synthesis.
- An SC_CTHREAD sensitive on the clock is used to model when the function is executed.
- The SC_CTHREAD supports reset behavior.

```
#include <systemc.h>

SC_MODULE(sc_sequ_cthread){
      //Ports
      sc_in <bool>  clk;
      sc_in <bool>  reset;
      sc_in <bool>  start;
      sc_in<sc_uint<16> > a;
      sc_in<bool> en;
      sc_out<sc_uint<16> > sum;
      sc_out<bool> vld;

      //Variables
      sc_uint<16> acc;

      //Process Declaration
      void accum();

      //Constructor
```

```
        SC_CTOR(sc_sequ_cthread){

                //Process Registration
                SC_CTHREAD(accum,clk.pos());
                reset_signal_is(reset,true);
        }
};
```

Example 4-65: **SystemC SC_CTHREAD Example**

Function `accum` is shown in Example 2-66. The important aspects highlighted by this examples are:

- The core modeling process is an infinite `while()` loop with a `wait()` statement inside it.
- Any initialization of the variables is performed before the infinite `while()` loop: this code is executed when reset is recognized by the `SC_CTHREAD`.
- The data reads and writes are qualified by handshake protocols.

```
#include "sc_sequ_cthread.h"

void sc_sequ_cthread::accum(){

      //Initialization
      acc=0;
      sum.write(0);
      vld.write(false);
      wait();

      // Process the data
      while(true) {
            // Wait for start
            while (!start.read()) wait();

            // Read if valid input available
            if (en) {
                  acc = acc + a.read();
                  sum.write(acc);
                  vld.write(true);
            } else {
                  vld.write(false);
            }
            wait();
      }

}
```

Example 4-66: **SystemC SC_CTHREAD Function**

Synthesis with Multiple Clocks

SystemC, unlike C and C++ synthesis, supports designs with multiple clocks. In a multiple clock design, the functionality associated with each clock must be captured in an `SC_CTHREAD`.

Example 2-67 shows a design in which there are two clocks, named `clock` and `clock2`. One is used to activate an `SC_CTHREAD` executing function `Prc1` and the other used to activate an `SC_CTHREAD` executing function `Prc2`. After synthesis, all the sequential logic associated with function `Prc1` will be clocked by clock, while `clock2` will drive all the sequential logic of function `Prc2`.

```
#include"systemc.h"
#include"tlm.h"
using namespace tlm;

SC_MODULE(sc_multi_clock)
{
     //Ports
     sc_in <bool>  clock;
     sc_in <bool>  clock2;
     sc_in <bool>  reset;
     sc_in <bool>  start;
     sc_out<bool>  done;
     sc_fifo_out<int> dout;
     sc_fifo_in<int> din;

     //Variables
     int share_mem[100];
     bool write_done;

     //Process Declaration
     void Prc1();
     void Prc2();

     //Constructor
     SC_CTOR(sc_multi_clock)
     {
            //Process Registration
            SC_CTHREAD(Prc1,clock.pos());
            reset_signal_is(reset,true);

            SC_CTHREAD(Prc2,clock2.pos());
            reset_signal_is(reset,true);
     }
};
```

Example 4-67: **SystemC Multiple Clock Design**

Top-Level SystemC Ports

The ports in a SystemC design are specified in the source code. The one major difference when using SystemC, as compared to C and C++ functions, is that Vivado HLS only performs interface synthesis on supported memory interfaces (refer to Arrays on the Interface). All port on the top-level interface must be of types `sc_in_clk`, `sc_in`, `sc_out`, `sc_inout`, `sc_fifo_in`, `sc_fifo_out` or `ap_mem_if`.

With the exception of the supported memory interfaces (`sc_fifo_in`, `sc_fifo_out` and `ap_mem_if`) all handshaking between the design and the test bench must be explicitly modeled in the SystemC function.

Note: Vivado HLS may add additional clock cycles to a SystemC design if this is required to meet timing. Since the number of clock cycles after synthesis may be different, SystemC designs should handshake all data transfers with the test bench.

Transaction level modeling using TLM 2.0 and event based modeling are not supported for synthesis.

SystemC Interface Synthesis

In general, Vivado HLS does not perform interface synthesis on SystemC. As mentioned above, it does support interface synthesis for some memory interfaces, namely RAM and FIFO ports.

RAM Port Synthesis

Unlike the synthesis of C and C++, Vivado HLS does not automatically transform array ports into RTL RAM ports. In the following SystemC code, Vivado HLS directives must be used to partition the array ports into individual elements or this example code cannot be synthesized:

```
SC_MODULE(dut)
{
      sc_in<T> in0[N];
      sc_out<T>out0[N];

      …
      SC_CTOR(dut)
      {
            …
      }
};
```

The directives which would partition these arrays into individual elements are:

```
set_directive_array_partition dut in0 -type complete
set_directive_array_partition dut out0 -type complete
```

If however N is a large number, this will result in many individual scalar ports on the RTL interface.

Example 2-68 shows how a RAM interface can be modeled in SystemC simulation and fully synthesized by Vivado HLS. In Example 2-68, the arrays are replaced by `ap_mem_if` types which can synthesized into RAM ports.

- To use `ap_mem_port` types, the header file `ap_mem_if.h` from the `include/ap_sysc` directory in the Vivado HLS installation area must be included.
  - Inside the Vivado HLS environment, the directory `include/ap_sysc` is automatically included.
- The arrays for din and `dout` are replaced by `ap_mem_port` types (The fields are explained after Example 2-68).

```
#include"systemc.h"
#include "ap_mem_if.h"


SC_MODULE(sc_RAM_port)
{
      //Ports
      sc_in <bool>  clock;
      sc_in <bool>  reset;
      sc_in <bool>  start;
      sc_out<bool>  done;
      //sc_out<int> dout[100];
      //sc_in<int> din[100];
      ap_mem_port<int, int, 100, RAM2P> dout;
      ap_mem_port<int, int, 100, RAM2P> din;

      //Variables
      int share_mem[100];
      sc_signal<bool> write_done;

      //Process Declaration
      void Prc1();
      void Prc2();

      //Constructor
      SC_CTOR(sc_RAM_port)
      : dout ("dout"),
      din ("din")
      {
            //Process Registration
            SC_CTHREAD(Prc1,clock.pos());
            reset_signal_is(reset,true);

            SC_CTHREAD(Prc2,clock.pos());
            reset_signal_is(reset,true);
      }
};
```

Example 4-68: **SystemC RAM Interface**

The format of the `ap_mem_port` type is:

```
ap_mem_port (<data_type>, < address_type>, <number_of_elements>, <Mem_Target>)
```

- The `data_type` is the type used for the stored data elements. In Example 2-68, these are standard int types.
- The `address_type` is the type used for the address bus. This type should have enough data bits to address all elements in the array, or C simulation will fail.
- The `number_of_elements` specifies the number of elements in the array being modeled.
- The `Mem_Target` specifies the memory to which this port will connect and hence determines the IO ports on the final RTL. A list of the available targets are provided in Table 2-11.

The memory targets described in Table 2-11 influence both the ports created by synthesis and how the operations are scheduled in the design. For example, a dual-port RAM will result in twice as many IO ports as a single-port RAM and may allow internal operations to be scheduled in parallel: if code constructs, such as loops, and data dependencies allow this.

Table 4-11: **System C ap_mem_port Memory Targets**

Target RAM	Description
RAM1P	A single-port RAM.
RAM2P	A dual-port RAM.
RAMT2P	A true dual-port RAM, with support for both read and write on both the input and output side
ROM1P	A single-port ROM.
ROM2P	A dual-port ROM.

Once the ap_mem_port has been defined on the interface, the variables are simply accessed in the code in the same manner as any other arrays:

```
dout[i] = share_mem[i] + din[i];
```

The test bench to support Example 2-68 is shown below in Example 2-69. The ap_mem_port type must be supported by an ap_mem_chn type in the test bench. The ap_mem_chn type is defined in the header file ap_mem_if.h and supports the same fields as ap_mem_port.

```
#ifdef __RTL_SIMULATION__
#include "sc_RAM_port_rtl_wrap.h"
#define sc_RAM_port sc_RAM_port_RTL_transactor
#else
#include "sc_RAM_port.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"
#include "ap_mem_if.h"

int sc_main (int argc , char *argv[])
{
 sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);

     sc_signal<bool>       s_reset;
     sc_signal<bool>       s_start;
     sc_signal<bool>       s_done;
     ap_mem_chn<int,int, 100, RAM2P> dout;
     ap_mem_chn<int,int, 100, RAM2P> din;

     // Create a 10ns period clock signal
     sc_clock s_clk("s_clk",10,SC_NS);

     tb_init      U_tb_init("U_tb_init");
```

```
    sc_RAM_port   U_dut("U_dut");
    tb_driver     U_tb_driver("U_tb_driver");

    // Generate a clock and reset to drive the sim
    U_tb_init.clk(s_clk);
    U_tb_init.reset(s_reset);
    U_tb_init.done(s_done);
    U_tb_init.start(s_start);

    // Connect the DUT
    U_dut.clock(s_clk);
    U_dut.reset(s_reset);
    U_dut.done(s_done);
    U_dut.start(s_start);
    U_dut.dout(dout);
    U_dut.din(din);

    // Drive inputs and Capture outputs
    U_tb_driver.clk(s_clk);
    U_tb_driver.reset(s_reset);
    U_tb_driver.start(s_start);
    U_tb_driver.done(s_done);
    U_tb_driver.dout(dout);
    U_tb_driver.din(din);

    // Sim
    int end_time = 1100;

    cout << "INFO: Simulating " << endl;

    // start simulation
    sc_start(end_time, SC_NS);

    if (U_tb_driver.retval != 0) {
          printf("Test failed  !!!\n");
    } else {
          printf("Test passed !\n");
  }
  return U_tb_driver.retval;
};
```

Example 4-69: **SystemC RAM Interface Test Bench**

FIFO Port Synthesis

FIFO ports on top-level interface can be synthesized directly from the standard SystemC `sc_fifo_in` and `sc_fifo_out` ports. An example on using FIFO ports on the interface is provided below (Example 2-70).

After synthesis each FIFO port will have a data port and associated FIFO control signals (inputs will have empty and read ports; outputs will have full and write ports). An advantage of using FIFO ports is that the handshake required to synchronize data transfers are automatically added in the RTL test bench.

```
#include"systemc.h"
#include"tlm.h"
```

```
using namespace tlm;


SC_MODULE(sc_FIFO_port)
{
      //Ports
      sc_in <bool>  clock;
      sc_in <bool>  reset;
      sc_in <bool>  start;
      sc_out<bool>  done;
      sc_fifo_out<int> dout;
      sc_fifo_in<int> din;

      //Variables
      int share_mem[100];
      bool write_done;

      //Process Declaration
      void Prc1();
      void Prc2();

      //Constructor
      SC_CTOR(sc_FIFO_port)
      {
            //Process Registration
            SC_CTHREAD(Prc1,clock.pos());
            reset_signal_is(reset,true);

            SC_CTHREAD(Prc2,clock.pos());
            reset_signal_is(reset,true);
      }
};
```

Example 4-70: **SystemC FIFO Interface**

Unsupported SystemC Constructs

Modules and Constructors

As mentioned above, but repeated here for reference, the following are not supported:

- An `SC_MODULE` cannot be nested inside another `SC_MODULE`.
- An `SC_MODULE` cannot be derived from another `SC_MODULE`.
- `SC_THREAD` is not supported (the clocked version, `SC_CTHREAD` is supported).

Instantiating Modules

An `SC_MODULE` cannot instantiated using new. Such code, `SC_MODULE(TOP)`

```
{
      sc_in<T> din;
      sc_out<T> dout;
```

```
        M1 *t0;

        SC_CTOR(TOP){
                t0 = new M1("t0");
                t0->din(din);
                t0->dout(dout);
        }
}
```

Must be transformed to:

```
SC_MODULE(TOP)
{
        sc_in<T> din;
        sc_out<T> dout;

        M1 t0;

        SC_CTOR(TOP)
        : t0("t0")
        {
                t0.din(din);
                t0.dout(dout);
        }
}
```

Module Constructors

Only name parameters can be used with module constructors. The following passing on variable temp of type int is not allowed.

```
SC_MODULE(dut) {
        sc_in<int> in0;
        sc_out<int>out0;
        int var;
        SC_HAS_PROCESS(dut);
        dut(sc_module_name nm, int temp)
:sc_module(nm),var(temp)
  { …   }
};
```

Functions

Virtual Functions are not supported. The following code cannot be synthesized due to the use of the virtual function.

```
SC_MODULE(DUT)
{
        sc_in<int> in0;
        sc_out<int>out0;

        virtual int foo(int var1)
        {
                  return var1+10;
        }
```

```
        void process()
    {
            int var=foo(in0.read());
            out0.write(var);
    }
    …
};
```

Top-Level Interface Ports

Reading an sc_out port is not supported. The following example is not allowed due to the read on `out0`.

```
SC_MODULE(DUT)
{
    sc_in<T> in0;
    sc_out<T>out0;
    …
    void process()
    {
    int var=in0.read()+out0.read();
    out0.write(var);
    }
};
```

C Arbitrary Precision Types

This section provides the details on the Arbitrary Precision (AP) types provided for C language design by Vivado HLS. Subsections provide details on the associated functions for C `int#w` types.

IMPORTANT: *When `[u]int#W` types are used, the `apcc` option must be selected in the project settings, to ensure the types are correctly simulated. Functions with these types cannot be analyzed in the debugger.*

Compiling [u]int#W Types

In order to use the `[u]int#W` types the `"ap_cint.h"` header file must be included in all source files which reference `[u]int#W` variables.

When compiling software models that use these types, it may be necessary to specify the location of the Vivado HLS header files, for example, by adding the `"-I/<HLS_HOME>/include"` option for `gcc` compilation.

Also note that best performance will be observed for software models when compiled with `gcc -O3` option.

Declaring/Defining [u]int#W Variables

There are separate signed and unsigned C types, respectively:

- `int#W`
- `uint#W`

The number `#W` specifies the total width of the variable being declared.

As usual, user defined types may be created with the C/C++ `'typedef'` statement as shown among the following examples:

```
include "ap_cint.h"              // use [u]int#W types

typedef uint128 uint128_t;       // 128-bit user defined type
int96 my_wide_var;               // a global variable declaration
```

The maximum width allowed is 1024 bits.

Initialization and Assignment from Constants (Literals)

A `[u]int#W` variable can be initialized with the same integer constants which are supported for the native integer data types. The constants will be zero or sign extended to the full width of the `[u]int#W` variable.

```
#include "ap_cint.h"

uint15      a      = 0;
uint52      b      = 1234567890U;
uint52      c      = 0o12345670UL;
uint96      d      = 0x123456789ABCDEFULL;
```

For bit-widths greater than 64-bit, the following functions can be used.

apint_string2bits()

This section also discusses use of the related functions:

- `apint_string2bits_bin()`
- `apint_string2bits_oct()`
- `apint_string2bits_hex()`

These functions convert a constant character string of digits, specified within the constraints of the radix (decimal, binary, octal, hexadecimal), into the corresponding value with the given bit-width `N`. For any radix, the number can be preceded with the minus sign -, to indicate a negative value.

```
int#Napint_string2bits[_radix](const char*, int N)
```

This is used to construct integer constants with values that are bigger than what the C language already permits. Smaller values work too, however are easier to specify with the existing C language constant value constructs:

```
#include <stdio.h>
#include "ap_cint.h"

int128 a;

// Set a to the value hex 00000000000000000123456789ABCDF0
a = a-apint_string2bits_hex("-123456789ABCDEF",128);
```

In addition, values can be assigned directly from a character string.

apint_vstring2bits()

This function converts a character string of digits, specified within the constraints of the hexadecimal radix, into the corresponding value with the given bit-width `N`. The number can be preceded with the minus sign -, to indicate a negative value.

This is used to construct integer constants with values that are larger than what the C language permits. The function is typically used in a test bench, to read information from a file.

Given file `test.dat` contains the following data:

```
123456789ABCDEF
-123456789ABCDEF
-5
```

The function, used in the test bench, would supply the following values:

```
#include <stdio.h>
#include "ap_cint.h"

typedef data_t;

int128 test (
     int128 t a
     ) {
     return a+1;
}

int main () {
     FILE *fp;
     char  vstring[33];

     fp    = fopen("test.dat","r");

     while (fscanf(fp,"%s",vstring)==1) {

            // Supply function "test" with the following values
            // 00000000000000000123456789ABCDF0
            // FFFFFFFFFFFFFFFFFEDCBA9876543212
            // FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
```

```
            test(apint_vstring2bits_hex(vstring,128));
            printf("\n");
        }

        fclose(fp);
        return 0;
}
```

Support for console I/O (Printing)

A `[u]int#W` variable can be printed with the same conversion specifiers that are supported for the native integer data types. Only the bits that fit according to the conversion specifier will be printed:

```
#include "ap_cint.h"

uint164                     c = 0x123456789ABCDEFULL;

printf("  d%40d\n",c);      // Signed integer in decimal format
// d                        -1985229329
printf(" hd%40hd\n",c);     // Short integer
// hd                       -12817
printf(" ld%40ld\n",c);     // Long integer
// ld                       81985529216486895
printf("lld%40lld\n",c);    // Long long integer
// lld                      81985529216486895

printf("  u%40u\n",c);      // Unsigned integer in decimal format
// u                        2309737967
printf(" hu%40hu\n",c);
// hu                       52719
printf(" lu%40lu\n",c);
// lu                       81985529216486895
printf("llu%40llu\n",c);
// llu                      81985529216486895

printf("  o%40o\n",c);      // Unsigned integer in octal format
// o                        21152746757
printf(" ho%40ho\n",c);
// ho                       146757
printf(" lo%40lo\n",c);
// lo                       4432126361152746757
printf("llo%40llo\n",c);
// llo                      4432126361152746757

printf("  x%40x\n",c);      // Unsigned integer in hexadecimal format [0-9a-f]
// x                        89abcdef
printf(" hx%40hx\n",c);
// hx                       cdef
printf(" lx%40lx\n",c);
// lx                       123456789abcdef
printf("llx%40llx\n",c);
// llx                      123456789abcdef

printf("  X%40X\n",c);      // Unsigned integer in hexadecimal format [0-9A-F]
// X                        89ABCDEF
```

```
}
```

As with initialization and assignment to `[u]int#W` variables, features are provided to support printing values which require more than 64-bits to represent.

apint_print()

This is used to print integers with values that are larger than what the C language already permits. This function prints a value to stdout, interpreted according to the radix (2, 8, 10, 16).

```
void         apint_print(int#N value, int radix)
```

The following example shows the results when `apint_printf()` is used:

```
#include <stdio.h>
#include "ap_cint.h"

int65 Var1 = 44;

apint_print(tmp,2);
//00000000000000000000000000000000000000000000000000000000000101100
apint_print(tmp,8);           // 0000000000000000054
apint_print(tmp,10);          // 44
apint_print(tmp,16);          // 0000000000000002C
```

apint_fprint()

This is used to print integers with values that are bigger than what the C language already permits. This function prints a value to a file, interpreted according to the radix (2, 8, 10, 16).

```
void        apint_fprint(FILE* file, int#N value, int radix)
```

Expressions Involving [u]int#W types

Variables of `[u]int#W` types may, for the most part, be used freely in expressions involving any C operators. However, there are some behaviors that may seem unexpected and bear detailed explanation.

Zero- and sign-extension on assignment from narrower to wider variables

When assigning the value of a narrower bit-width signed variable to a wider one, the value will be sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable will be zero-extended before assignment.

Explicit casting of the source variable may be necessary in order to ensure expected behavior on assignment.

Truncation on assignment of wider to narrower variables

Assigning a wider source variables value to a narrower one will lead to truncation of the value, with all bits beyond the most significant bit (MSB) position of the destination variable being lost.

There is no special handling of the sign information during truncation, which may lead to unexpected behavior. Again, explicit casting may help avoid unexpected behavior.

Binary Arithmetic Operators

In general, any valid operation that may be done on a native C integer data type, is supported for `[u]int#w` types.

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. All of the following operators take either two operands of `[u]int#W` or one `[u]int#W` type and one C/C++ fundamental integer data type, e.g., `char`, `short`, `int`, etc.

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

Note that when expressions contain a mix of `ap_[u]int` and C/C++ fundamental integer types, the C++ types will assume the following widths:

- `char`: 8-bits
- `short`: 16-bits
- `int`: 32-bits
- `long`: 32-bits
- `long long`: 64-bits

Addition

```
[u]int#W::RType [u]int#W::operator + ([u]int#W op)
```

This operator produces the sum of two `ap_[u]int (or one ap_[u]int` and a C/C++ integer type).

The width of the sum value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower is signed).

The sum will be treated as signed if either (or both) of the operands is of a signed type.

Subtraction

```
[u]int#W::RType [u]int#W::operator - ([u]int#W op)
```

This operator produces the difference of two integers.

The width of the difference value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower signed), before assignment, at which point it will be sign-extended, zero-padded or truncated based on the width of the destination variable.

The difference will be treated as signed regardless of the signedness of the operands.

Multiplication

```
[u]int#W::RType [u]int#W::operator * ([u]int#W op)
```

This operator returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product will be treated as a signed type if either of the operands is of a signed type.

Division

```
[u]int#W::RType [u]int#W::operator / ([u]int#W op)
```

This operator returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type; otherwise it is the width of the dividend plus one.

The quotient will be treated as a signed type if either of the operands is of a signed type.

Note: Vivado HLS synthesis of the divide operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE™ IP divider core(s) in the generated RTL.

Modulus

```
[u]int#W::RType [u]int#W::operator % ([u]int#W op)
```

This operator returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness; if the divisor is an unsigned type and the dividend is signed then the width is that of the divisor plus one.

The quotient will be treated as having the same signedness as the dividend.

Note: Vivado HLS synthesis of the modulus (%) operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands and will be treated as unsigned if and only if both operands are unsigned, otherwise it will be of a signed type.

Note that sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR

```
[u]int#W::RType [u]int#W::operator | ([u]int#W op)
```

Returns the bitwise OR of the two operands.

Bitwise AND

```
[u]int#W::RType [u]int#W::operator & ([u]int#W op)
```

Returns the bitwise AND of the two operands.

Bitwise XOR

```
[u]int#W::RType [u]int#W::operator ^ ([u]int#W op)
```

Returns the bitwise XOR of the two operands.

Shift Operators

Each shift operator comes in two versions, one for unsigned right-hand side (RHS) operands and one for signed RHS.

A negative value supplied to the signed RHS versions reverses the shift operations direction, i.e. a shift by the absolute value of the RHS operand in the opposite direction will occur.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit will be copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right

```
[u]int#W [u]int#W::operator << (ap_uint<int_W2> op)
```

Integer Shift Right

```
[u]int#W [u]int#W::operator << (ap_int<int_W2> op)
```

Unsigned Integer Shift Left

```
[u]int#W [u]int#W::operator >> (ap_uint<int_W2> op)
```

Integer Shift Left

```
[u]int#W [u]int#W::operator >> (ap_int<int_W2> op)
```

Beware when assigning the result of a shift-left operator to a wider destination variable, as some (or all) information may be lost. It is recommended to explicitly cast the shift expression to the destination type in order to avoid unexpected behavior.

Compound Assignment Operators

The compound assignment operators are supported:

```
*=  /=  %=  +=  -=  <<= >>=  &=  ^=  |=
```

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness and potential sign-extension or truncation rules apply as detailed above for the relevant operations.

Relational Operators

All relational operators are supported and return a Boolean value based on the result of the comparison. Variables of `ap_[u]int` types may be compared to C/C++ fundamental integer types with these operators.

Equality

```
bool [u]int#W::operator == ([u]int#W op)
```

Inequality

```
bool [u]int#W::operator != ([u]int#W op)
```

Less than

```
bool [u]int#W::operator < ([u]int#W op)
```

Greater than

```
bool [u]int#W::operator > ([u]int#W op)
```

Less than or equal

```
bool [u]int#W::operator <= ([u]int#W op)
```

Greater than or equal

```
bool [u]int#W::operator >= ([u]int#W op)
```

Bit-Level Operation: Support Function

The `[u]int#W` types allow variables to be expressed with bit-level accuracy. It is often desirable with (hardware) algorithms that bit-level operations be performed. Vivado HLS provides the following functions to enable this.

Bit Manipulation

The following methods are provided in order to facilitate common bit-level operations on the value stored in `ap_[u]int` type variable(s).

Length

apint_bitwidthof()

```
int    apint_bitwidthof(type_or_value)
```

This function returns an integer value that provides the number of bits in an arbitrary precision integer value; It can be used with a type or a value:

```
int5 Var1, Res1;

Var1= -1;
Res1 = apint_bitwidthof(Var1);    // Res1 is assigned 5
Res1 = apint_and_reduce(int7);    // Res1 is assigned 7
```

Concatenation

apint_concatenate()

```
int#(N+M)    apint_concatenate(int#N first, int#M second)
```

Concatenates two `[u]int#W` variables, the width of the returned value is the sum of the widths of the operands.

The high and low arguments will be placed in the higher and lower order bits of the result respectively.

C native types, including integer literals, should be explicitly cast to an appropriate `[u]int#W` type before concatenating in order to avoid unexpected results.

Bit selection

apint_get_bit()

```
int    apint_get_bit(int#N source, int index)
```

This operation function selects one bit from an arbitrary precision integer value and returns it.

The source must be an `[u]int#W` type and the index argument must be an int value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `[u]int#W`.

Set bit value

apint_set_bit()

```
int#N    apint_set_bit(int#N source, int index, int value)
```

This function sets the specified bit, index, of the `[u]int#W` instance source to the value specified (zero or one).

Range selection

apint_get_range()

```
int#N    apint_get_range(int#N source, int high, int low)
```

This operation returns the value represented by the range of bits specified by the arguments.

The `Hi` argument specifies the most significant bit (MSB) position of the range and `Lo` the least significant (LSB).

The LSB of the source variable is in position 0. If the `Hi` argument has a value less than `Lo`, then the bits are returned in reverse order.

Set range value

apint_set_range()

```
int#N       apint_set_range(int#N source, int high, int low, int#M part)
```

This function sets the bits specified of source between, `high` and `low`, to the value of `part`.

Bit Reduction

AND reduce

apint_and_reduce()

```
int         apint_and_reduce(int#N value)
```

This function applies the `AND` operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool):

```
int5 Var1, Res1;

Var1= -1;
Res1 = apint_and_reduce(Var1);    // Res1 is assigned 1
```

```
Var1= 1;
Res1 = apint_and_reduce(Var1);    // Res1 is assigned 0
```

This operation is equivalent go comparing to -1, and return a 1 if it matches, 0 otherwise. Another interpretation is to check that all bits are one.

OR reduce

apint_or_reduce()

```
int             apint_or_reduce(int#N value)
```

This function applies the OR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go comparing to 0, and return a 0 if it matches, 1 otherwise.

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_or_reduce(Var1);     // Res1 is assigned 1

Var1= 0;
Res1 = apint_or_reduce(Var1);     // Res1 is assigned 0
```

XOR reduce

apint_xor_reduce()

```
int              apint_xor_reduce(int#N value)
```

This function applies the OR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go counting the ones in the word, and return 1 if there are an even number, or 0 if there are an odd number (even parity).

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_xor_reduce(Var1);       // Res1 is assigned 0

Var1= 0;
Res1 = apint_xor_reduce(Var1);       // Res1 is assigned 1
```

NAND reduce

apint_nand_reduce()

```
int               apint_nand_reduce(int#N value)
```

This function applies the NAND operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This is equivalent to comparing this value against -1 (all ones) and returning false if it matches, true otherwise.

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_nand_reduce(Var1);      // Res1 is assigned 1

Var1= -1;
Res1 = apint_nand_reduce(Var1);      // Res1 is assigned 0
```

NOR reduce

apint_nor_reduce()

```
int              apint_nor_reduce(int#N value)
```

This function applies the NOR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This is equivalent to comparing this value against 0 (all zeros) and returning true if it matches, false otherwise.

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_nor_reduce(Var1);       // Res1 is assigned 1

Var1= 1;
Res1 = apint_nor_reduce(Var1);       // Res1 is assigned 0
```

XNOR reduce

apint_xnor_reduce()

```
int               apint_xnor_reduce(int#N value)
```

This function applies the XNOR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go counting the ones in the word, and return 1 if there are an odd number, or 0 if there are an even number (odd parity).

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_xnor_reduce(Var1);      // Res1 is assigned 0

Var1= 1;
Res1 = apint_xnor_reduce(Var1);      // Res1 is assigned 1
```

C++ Arbitrary Precision Types

Vivado HLS provides a C++ template class, ap_[u]int<>, that implements arbitrary precision (or bit-accurate) integer data types with consistent, bit-accurate behavior between software and hardware modeling.

This class provides all arithmetic, bit-wise, logical and relational operators allowed for native C integer types. In addition this class provides methods to handle some useful hardware operations, such as allowing initialization and conversion of variables of widths greater than 64 bits. Details for all operators and class methods are detailed below.

Compiling ap_[u]<> Types

In order to use the `ap_[u]fixed<>` classes one must include the `"ap_int.h"` header file in all source files which reference `ap_[u]fixed<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the Vivado HLS header files, for example by adding the `"-I/<HLS_HOME>/include"` option for `g++` compilation.

Also note that best performance will be observed for software models when compiled with `g++ -O3` option.

Declaring/Defining ap_[u] Variables

There are separate signed and unsigned classes: `ap_int<int_W> & ap_uint<int _W>` respectively. The template parameter `int_W` specifies the total width of the variable being declared.

As usual, user defined types may be created with the C/C++ `'typedef'` statement as shown among the following examples:

```
include "ap_int.h"//                 use ap_[u]fixed<> types

typedef ap_uint<128> uint128_t;      // 128-bit user defined type
ap_int<96> my_wide_var;              // a global variable declaration
```

The default maximum width allowed is 1024 bits; this default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 32768 before inclusion of the `"ap_int.h"` header file.

CAUTION! *Setting the value of `AP_INT_MAX_W` too high may cause slow software compile and run times.*

Example of overriding `AP_INT_MAX_W`:

```
#define AP_INT_MAX_W 4096          // Must be defined before next line
#include "ap_int.h"

ap_int<2048> very_wide_var;
```

Initialization and Assignment from Constants (Literals)

The class constructor and assignment operator overloads, allows initialization of and assignment to `ap_[u]fixed<>` variables using standard C/C++ integer literals.

However, this method of assigning values to `ap_[u]fixed<>` variables is subject to the limitations of C++ and the system upon which the software will run, typically leading to a 64-bit limit on integer literals (e.g. for those `LL` or `ULL` suffixes).

In order to allow assignment of values wider than 64-bits, the `ap_[u]fixed<>` classes provide constructors that allow initialization from a string of arbitrary length (less than or equal to the width of the variable).

By default, the string provided will be interpreted as a hexadecimal value as long as it contains only valid hexadecimal digits (i.e. 0-9 and a-f). In order to assign a value from such a string, an explicit C++ style cast of the string to the appropriate type must be made.

Examples of initialization and assignments, including for values greater than 64-bit, are:

```
ap_int<42> a_42b_var(-1424692392255LL);        // long long decimal format
a_42b_var = 0x14BB648B13FLL;                   // hexadecimal format

a_42b_var = -1;            // negative int literal sign-extended to full width

ap_uint<96> wide_var("76543210fedcba9876543210");   // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567");
```

The `ap_[u]<>` constructor may be explicitly instructed to interpret the string as representing the number in radix 2, 8, 10, or 16 formats. This is accomplished by adding the appropriate radix value as a second parameter to the constructor call.

If the string literal provided contains any characters that are invalid as digits for the radix specified a compilation error will occur.

Examples using different radix formats:

```
ap_int<6> a_6bit_var("101010", 2);   // 42d in binary format
a_6bit_var = ap_int<6>("40", 8);     // 32d in octal format
a_6bit_var = ap_int<6>("55", 10);    // decimal format
a_6bit_var = ap_int<6>("2A", 16);    // 42d in hexadecimal format

a_6bit_var = ap_int<6>("42", 2);   // COMPILE-TIME ERROR! "42" is not binary
```

The radix of the number encoded in the string can also be inferred by the constructor, when it is prefixed with a zero (`0`) followed by one of the following characters: "`b`", "`o`" or "`x`"; the prefixes "`0b`", "`0o`" and "`0x`" correspond to binary, octal and hexadecimal formats respectively.

Examples using alternate initializer string formats:

```
ap_int<6> a_6bit_var("0b101010", 2);   // 42d in binary format
a_6bit_var = ap_int<6>("0o40", 8);     // 32d in octal format
```

```
a_6bit_var = ap_int<6>("0x2A", 16);    // 42d in hexidecimal format

a_6bit_var = ap_int<6>("0b42", 2);     // COMPILE-TIME ERROR! "42" is not binary
```

Support for console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, features are provided to support printing values which require more than 64-bits to represent.

The easiest way to output any value stored in an `ap_[u]int` variable is to use the C++ standard output stream, `std::cout (#include <iostream> or <iostream.h>)`. The stream insertion operator, "<<", is overloaded to correctly output the full range of values possible for any given `ap_[u]int` variable. The stream manipulators "`dec`", "`hex`" and "`oct`" are also supported, allowing formatting of the value as decimal, hexadecimal or octal respectively.

Example using "`cout`" to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_uint<72> Val("10fedcba9876543210");

cout << Val << endl;         // Yields: "313512663723845890576"
cout << hex << val << endl;  // Yields: "10fedcba9876543210"
cout << oct << val << endl;  // Yields: "417733456514166250310 20"
```

It is also possible to use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits, by first converting the value to a C++ std::string, then to a C character string. The `ap_[u]int` classes provide a method, `to_string()` to do the first conversion and the `std::string` class provides the `c_str()` method to convert to a null-terminated character string.

The ap[u]int::to_string() method may be passed an optional argument specifying the radix of the numerical format desired. The valid radix argument values are 2, 8, 10 & 16 for binary, octal, decimal and hexadecimal respectively; the default radix value is 16.

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean and the default value is false, causing the non-decimal formats to be printed as unsigned values.

Examples for using `printf` to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str();        // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str();      // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str();       // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str();   // => "-7F0123456789ABCDF0"
```

Expressions Involving ap_[u]<> types

Variables of `ap_[u]<>` types may, for the most part, be used freely in expressions involving any C/C++ operators. However, there are some behaviors that may seem unexpected and bear detailed explanation.

Zero- and sign-extension on assignment from narrower to wider variables

When assigning the value of a narrower bit-width) signed (`ap_int<>`) variable to a wider one, the value will be sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable will be zero-extended before assignment.

Explicit casting of the source variable, as shown below, may be necessary in order to ensure expected behavior on assignment.

```
ap_uint<10> Result;

ap_int<7> Val1 = 0x7f;
ap_uint<6> Val2 = 0x3f;

Result = Val1;                // Yields: 0x3ff (sign-extended)
Result = Val2;                // Yields: 0x03f (zero-padded)

Result = ap_uint<7>(Val1);    // Yields: 0x07f (zero-padded)
Result = ap_int<6>(Val2);     // Yields: 0x3ff (sign-extended)
```

Truncation on assignment of wider to narrower variables

Assigning a wider source variables value to a narrower one will lead to truncation of the value, with all bits beyond the most significant bit (MSB) position of the destination variable being lost.

There is no special handling of the sign information during truncation, which may lead to unexpected behavior. Again, explicit casting may help avoid unexpected behavior.

Class Operators and Methods

In general, any valid operation that may be done on a native C/C++ integer data type, is supported, via operator overloading, for `ap_[u]int` types.

In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. All of the following operators take either two operands of `ap_[u]int` or one

ap_[u]int type and one C/C++ fundamental integer data type, e.g. char, short, int, etc.

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

Note that when expressions contain a mix of ap_[u]int and C/C++ fundamental integer types, the C++ types will assume the following widths:

- char: 8-bits
- short: 16-bits
- int: 32-bits
- long: 32-bits
- long long: 64-bits

Addition

```
ap_(u)int::RType ap_(u)int::operator + (ap_(u)int op)
```

This operator produces the sum of two ap_[u]int (or one ap_[u]int and a C/C++ integer type).

The width of the sum value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower is signed).

The sum will be treated as signed if either (or both) of the operands is of a signed type.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - (ap_(u)int op)
```

This operator produces the difference of two integers.

The width of the difference value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower signed), before assignment, at which point it will be sign-extended, zero-padded or truncated based on the width of the destination variable.

The difference will be treated as signed regardless of the signedness of the operands.

Multiplication

```
ap_(u)int::RType ap_(u)int::operator * (ap_(u)int op)
```

This operator returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product will be treated as a signed type if either of the operands is of a signed type.

Division

```
ap_(u)int::RType ap_(u)int::operator / (ap_(u)int op)
```

This operator returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type; otherwise it is the width of the dividend plus one.

The quotient will be treated as a signed type if either of the operands is of a signed type.

IMPORTANT: *Vivado HLS synthesis of the divide operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.*

Modulus

```
ap_(u)int::RType ap_(u)int::operator % (ap_(u)int op)
```

This operator returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness; if the divisor is an unsigned type and the dividend is signed then the width is that of the divisor plus one.

The quotient will be treated as having the same signedness as the dividend.

IMPORTANT: *Vivado HLS synthesis of the modulus (%) operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.*

Examples of arithmetic operators:

```
ap_uint<71> Rslt;

ap_uint<42> Val1 = 5;
ap_int<23> Val2 = -8;

Rslt = Val1 + Val2;     // Yields: -3 (43 bits) sign-extended to 71 bits
Rslt = Val1 - Val2;     // Yields: +3 sign extended to 71 bits
Rslt = Val1 * Val2;     // Yields: -40 (65 bits) sign extended to 71 bits
Rslt = 50 / Val2;       // Yields: -6 (33 bits) sign extended to 71 bits
Rslt = 50 % Val2;       // Yields: +2 (23 bits) sign extended to 71 bits
```

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands and will be treated as unsigned if and only if both operands are unsigned, otherwise it will be of a signed type.

Note that sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR

```
ap_(u)int::RType ap_(u)int::operator | (ap_(u)int op)
```

Returns the bitwise OR of the two operands.

Bitwise AND

```
ap_(u)int::RType ap_(u)int::operator & (ap_(u)int op)
```

Returns the bitwise AND of the two operands.

Bitwise XOR

```
ap_(u)int::RType ap_(u)int::operator ^ (ap_(u)int op)
```

Returns the bitwise XOR of the two operands.

Unary Operators

Addition

```
ap_(u)int ap_(u)int::operator + ()
```

Returns the self copy of the `ap_[u]int` operand.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - ()
```

This operator returns the negated value of the operand with the same width if it is a signed type or its width plus one if it is unsigned.

The return value is always a signed type.

Bit-wise Inverse

```
ap_(u)int::RType ap_(u)int::operator ~ ()
```

This operator returns the bitwise-NOT of the operand with the same width and signedness.

Equality Zero

```
bool ap_(u)int::operator ! ()
```

This operator returns a Boolean "`false`" value if and only if the operand is equal to zero (`0`), "`true`" otherwise.

Shift Operators

Each shift operator comes in two versions, one for unsigned right-hand side (RHS) operands and one for signed RHS.

A negative value supplied to the signed RHS versions reverses the shift operations direction, i.e., a shift by the absolute value of the RHS operand in the opposite direction will occur.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit will be copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right

```
ap_(u)int ap_(u)int::operator << (ap_uint<int_W2> op)
```

Integer Shift Right

```
ap_(u)int ap_(u)int::operator << (ap_int<int_W2> op)
```

Unsigned Integer Shift Left

```
ap_(u)int ap_(u)int::operator >> (ap_uint<int_W2> op)
```

Integer Shift Left

```
ap_(u)int ap_(u)int::operator >> (ap_int<int_W2> op)
```

CAUTION! *Beware when assigning the result of a shift-left operator to a wider destination variable, as some (or all) information may be lost. It is recommended to explicitly cast the shift expression to the destination type in order to avoid unexpected behavior.*

Example for shift operations:

```
ap_uint<13> Rslt;

ap_uint<7> Val1 = 0x41;

Rslt = Val1 << 6;                 // Yields: 0x0040, i.e. msb of Val1 is lost
Rslt = ap_uint<13>(Val1) << 6;    // Yields: 0x1040, no info lost

ap_int<7> Val2 = -63;
Rslt = Val2 >> 4;                 //Yields: 0x1ffc, sign is maintained and extended
```

Compound Assignment Operators

The compound assignment operators are supported:

```
*=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
```

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness and potential sign-extension or truncation rules apply as detailed above for the relevant operations.

Example of a compound assignment statement:

```
ap_uint<10> Val1 = 630;
ap_int<3> Val2 = -3;
ap_uint<5> Val3 = 27;

Val1 += Val2 - Val3;          // Yields: 600 and is equivalent to:

// Val1 = ap_uint<10>(ap_int<11>(Val1) +
//        ap_int<11>((ap_int<6>(Val2) -
//        ap_int<6>(Val3))));
```

Increment & Decrement Operators

The increment and decrement operators are provided. All return a value of the same width as the operand and which is unsigned if and only if both operands are of unsigned types and signed otherwise.

Pre-increment

```
ap_(u)int& ap_(u)int::operator ++ ()
```

This operator returns the incremented value of the operand as well as assigning the incremented value to the operand.

Post-increment

```
const ap_(u)int ap_(u)int::operator ++ (int)
```

This operator returns the value of the operand before assignment of the incremented value to the operand variable.

Pre-decrement

```
ap_(u)int& ap_(u)int::operator -- ()
```

This operator returns the decremented value of, as well as assigning the decremented value to, the operand.

Post-decrement

```
const ap_(u)int ap_(u)int::operator -- (int)
```

This operator returns the value of the operand before assignment of the decremented value to the operand variable.

Relational Operators

All relational operators are supported and return a Boolean value based on the result of the comparison. Variables of `ap_[u]int` types may be compared to C/C++ fundamental integer types with these operators.

Equality

```
bool ap_(u)int::operator == (ap_(u)int op)
```

Inequality

```
bool ap_(u)int::operator != (ap_(u)int op)
```

Less than

```
bool ap_(u)int::operator < (ap_(u)int op)
```

Greater than

```
bool ap_(u)int::operator > (ap_(u)int op)
```

Less than or equal

```
bool ap_(u)int::operator <= (ap_(u)int op)
```

Greater than or equal

```
bool ap_(u)int::operator >= (ap_(u)int op)
```

Other Class Methods and Operators

Bit-level Operations

The following methods are provided in order to facilitate common bit-level operations on the value stored in `ap_[u]int` type variable(s).

Length

```
int ap_(u)int::length ()
```

This method returns an integer value providing the total number of bits in the `ap_[u]int` variable.

Concatenation

```
ap_concat_ref ap_(u)int::concat (ap_(u)int low)
ap_concat_ref ap_(u)int::operator , (ap_(u)int high, ap_(u)int low)
```

Concatenates two `ap_[u]int` variables, the width of the returned value is the sum of the widths of the operands.

The high and low arguments will be placed in the higher and lower order bits of the result respectively; the `concat()` method places the argument in the lower order bits.

When using the overloaded comma operator, the parentheses are required. The comma operator version may also appear on the LHS of assignment.

C/C++ native types, including integer literals, should be explicitly cast to an appropriate `ap_[u]int` type before concatenating in order to avoid unexpected results.

Examples of concatenation:

```
ap_uint<10> Rslt;

ap_int<3> Val1 = -3;
ap_int<7> Val2 = 54;

Rslt = (Val2, Val1);       // Yields: 0x1B5
Rslt = Val1.concat(Val2);  // Yields: 0x2B6
(Val1, Val2) = 0xAB;       // Yields: Val1 == 1, Val2 == 43
```

Bit selection

```
ap_bit_ref ap_(u)int::operator [] (int bit)
```

This operation function selects one bit from an arbitrary precision integer value and returns it.

The returned value is a reference value, which can be used to set or clear the corresponding bit in this `ap_[u]int`.

The bit argument must be an int value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]int`.

The result type `ap_bit_ref` represents the reference to one bit of this `ap_[u]int` instance specified by bit.

Range selection

```
ap_range_ref ap_(u)int::range (unsigned Hi, unsigned Lo)
ap_range_ref ap_(u)int::operator () (unsigned Hi, unsigned Lo)
```

This operation returns the value represented by the range of bits specified by the arguments.

The Hi argument specifies the most significant bit (MSB) position of the range and Lo the least significant (LSB).

The LSB of the source variable is in position 0. If the Hi argument has a value less than Lo, then the bits are returned in reverse order.

Examples using range selection:

```
ap_uint<4> Rslt;

ap_uint<8> Val1 = 0x5f;
ap_uint<8> Val2 = 0xaa;

Rslt = Val1.range(3, 0);   // Yields: 0xF
Val1(3,0) = Val2(3, 0);    // Yields: 0x5A
Val1(4,1) = Val2(4, 1);    // Yields: 0x55
Rslt = Val1.range(7, 4);   // Yields: 0xA; bit-reversed!
```

AND reduce

```
bool ap_(u)int::and_reduce ()
```

This operation function applies the AND operation on all bits in this ap_(u)int and returns the resulting single bit. This is equivalent to comparing this value against -1 (all ones) and returning true if it matches, false otherwise.

OR reduce

```
bool ap_(u)int::or_reduce ()
```

This operation function applies the OR operation on all bits in this ap_(u)int and returns the resulting single bit. This is equivalent to comparing this value against 0 (all zeros) and returning false if it matches, true otherwise.

XOR reduce

```
bool ap_(u)int::xor_reduce ()
```

This operation function applies the XOR operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to counting the number of 1 bits in this value and returning false if the count is even or true if the count is odd.

NAND reduce

```
bool ap_(u)int::nand_reduce ()
```

This operation function applies the NAND operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to comparing this value against -1 (all ones) and returning false if it matches, true otherwise.

NOR reduce

```
bool ap_int::nor_reduce ()
```

This operation function applies the NOR operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to comparing this value against 0 (all zeros) and returning true if it matches, false otherwise.

XNOR reduce

```
bool ap_(u)int::xnor_reduce ()
```

This operation function applies the XNOR operation on all bits in this ap_(u)int and returns the resulting single bit. This is equivalent to counting the number of 1 bits in this value and returning true if the count is even or false if the count is odd.

Examples of the various bit reduction methods:

```
ap_uint<8> Val = 0xaa;

bool t = Val.and_reduce(); // Yields: false
t = Val.or_reduce();       // Yields: true
t = Val.xor_reduce();      // Yields: false
t = Val.nand_reduce();     // Yields: true
t = Val.nor_reduce();      // Yields: false
t = Val.xnor_reduce();     // Yields: true
```

Bit reverse

```
void ap_(u)int::reverse ()
```

This member function reverses the contents of ap_[u]int instance, i.e. the LSB becomes the MSB and vice versa.

Example of reverse method:

```
ap_uint<8> Val = 0x12;

Val.reverse(); // Yields: 0x48
```

Test bit value

```
bool ap_(u)int::test (unsigned i)
```

This member function check whether specified bit of ap_(u)int instance is 1, returning true if so, false otherwise.

Example of test method:

```
ap_uint<8> Val = 0x12;

bool t = Val.test(5); // Yields: true
```

Set bit value

```
void ap_(u)int::set (unsigned i, bool v)
void ap_(u)int::set_bit (unsigned i, bool v)
```

This member function sets the specified bit of the `ap_(u)int` instance to the value of integer `v`.

Set bit (to 1)

```
void ap_(u)int::set (unsigned i)
```

This member function setVivado Design Suite User Guides the specified bit of the `ap_(u)int` instance to the value `1` (one).

Clear bit (to 0)

```
void ap_(u)int:: clear(unsigned i)
```

This member function sets the specified bit of the `ap_(u)int` instance to the value `0` (zero).

Invert bit

```
void ap_(u)int:: invert(unsigned i)
```

This member function inverts *i*th bit of the `ap_(u)int` instance, i.e. the *i*th bit will become `0` if its original value is `1` and vice versa.

Example of bit set, clear and invert bit methods:

```
ap_uint<8> Val = 0x12;
Val.set(0, 1);           // Yields: 0x13
Val.set_bit(5, false);   // Yields: 0x03
Val.set(7);              // Yields: 0x83
Val.clear(1);            // Yields: 0x81
Val.invert(5);           // Yields: 0x91
```

Rotate Right

```
void ap_(u)int:: rrotate(unsigned n)
```

This member function rotate the `ap_(u)int` instance *n* places to right.

Rotate Left

```
void ap_(u)int:: lrotate(unsigned n)
```

This member function rotate the `ap_(u)int` instance *n* places to left.

Examples of rotate methods:

```
ap_uint<8> Val = 0x12;

Val.rrotate(3);     // Yields: 0x42
Val.lrotate(6);     // Yields: 0x90
```

Bitwise NOT

```
void ap_(u)int:: b_not()
```

This member function complements every bit of the `ap_(u)int` instance.

Example:

```
ap_uint<8> Val = 0x12;

Val.b_not();   // Yields: 0xED
```

Test sign

```
bool ap_int:: sign()
```

This member function checks whether the `ap_(u)int` instance is negative, returning `true` if negative and return `false` if positive.

Explicit Conversion Methods

To C/C++ "(u)int"

```
int ap_(u)int::to_int ()
unsigned ap_(u)int::to_uint ()
```

These methods return native C/C++ (32-bit on most systems) integers with the value contained in the `ap_[u]int`. If the value is greater than can be represented by an "`[unsigned] int`", then truncation will occur.

To C/C++ 64-bit "(u)int"

```
long long ap_(u)int::to_int64 ()
unsigned long long ap_(u)int::to_uint64 ()
```

These methods return native C/C++ 64-bit integers with the value contained in the `ap_[u]int`. If the value is greater than can be represented by an "`[unsigned] int`", then truncation will occur.

To C/C++ "double"

```
double ap_(u)int::to_double ()
```

This method returns a native C/C++ "double" 64-bit floating point representation of the value contained in the `ap_[u]int`. Note that if the `ap_[u]int` is wider than 53 bits (the number of bits in the mantissa of a "`double`"), the resulting "`double`" may not have the exact value expected.

C++ Arbitrary Precision Fixed Point Types

Vivado HLS provides support for fixed point types which allow fractional arithmetic to be easily handled. The advantage of fixed point arithmetic is shown in the following example.

```
ap_fixed<10, 5> Var1 = 22.96875;        // 10-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5           // 12-bit word, 1 fractional bit
ap_fixed<13,5> Res1;                    // 13-bit signed word, 5 fractional bits

Res1 = Var1 + Var2;                     // Result is 535.46875
```

Even though `Var1` and `Var2` have different precisions, the fixed point type ensures the decimal point is correctly aligned before the operation (an addition in this case), is performed. The user is not required to perform any operations in the C code to align the decimal point.

The type used to store the result of any fixed point arithmetic operation must be large enough, in both the integer and fractional bits, to store the full result.

If this is not the case, the `ap_fixed` type automatically performs overflow handling (when the result has more MSBs than the assigned type supports) and quantization (or rounding: when the result has fewer LSBs than the assigned type supports). The `ap_[u]fixed` type provides a number of options, detailed below, on how the overflow and quantization are performed.

The ap_[u]fixed Representation

In `ap_[u]fixed` types, a fixed-point value is represented as a sequence of bits with a specified position for the binary point. Bits to the left of the binary point represent the integer part of the value and bits to the right of the binary point represent the fractional part of the value.

`ap_[u]fixed type` is defined as follows:

```
ap_[u]fixed<int W,
            int I,
            ap_q_mode Q,
            ap_o_mode O,
            ap_sat_bits N>;
```

- The `W` attribute takes one parameter: the total number of bits for the word. Only a constant integer expression can be used as the parameter value.
- The `I` attribute takes one parameter: the number of bits to represent the integer part. The value of `I` must be less than or equal to `W`. The number of bits to represent the fractional part is `W` minus I. Only a constant integer expression can be used as the parameter value.

- The Q attribute takes one parameter: quantization mode. Only predefined enumerated value can be used as the parameter value. The default value is AP_TRN.
- The O attribute takes one parameter: overflow mode. Only predefined enumerated value can be used as the parameter value. The default value is AP_WRAP.
- The N attribute takes one parameter: the number of saturation bits considered used in the overflow wrap modes. Only a constant integer expression can be used as the parameter value. The default value is zero.

Note: If the quantization, overflow and saturation parameters are not specified, as in the first example above, the default settings are used.

The quantization and overflow modes are explained below.

Quantization modes

Rounding to plus infinity	AP_RND
Rounding to zero	AP_RND_ZERO
Rounding to minus infinity	AP_RND_MIN_INF
Rounding to infinity	AP_RND_INF
Convergent rounding	AP_RND_CONV
Truncation	AP_TRN
Truncation to zero	AP_TRN_ZERO

AP_RND

The AP_RND quantization mode indicates that the value should be rounded to the nearest representable value for the specific ap_[u]fixed type.

For example:

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25;      // Yields: 1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25;     // Yields: -1.0
```

AP_RND_ZERO

The AP_RND_ZERO quantization mode indicates the value should be rounded to the nearest representable value and rounding should be towards zero. That is, for positive value, the redundant bits should be deleted, while for negative value, the least significant bits should be added to get the nearest representable value.

For example:

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25;  // Yields: 1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_RND_MIN_INF

The AP_RND_MIN_INF quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding should be towards minus infinity. That is, for positive value, the redundant bits should be deleted, while for negative value, the least significant bits should be added.

For example:

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25;   // Yields: -1.5
```

AP_RND_INF

The AP_RND_INF quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding depends on the least significant bit.

* For positive values, if the least significant bit is set, round towards plus infinity, otherwise, round towards minus infinity.

* For negative value, if the least significant bit is set, round towards minus infinity, otherwise, round towards plus infinity.

For example:

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25;     // Yields: 1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25;    // Yields: -1.5
```

AP_RND_CONV

The AP_RND_CONV quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding depends on the least significant bit. If the least significant bit is set, round towards plus infinity, otherwise, round towards minus infinity.

For example:

```
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = 0.75;    // Yields: 1.0
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = -1.25;   // Yields: -1.0
```

AP_TRN

The AP_TRN quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding should always towards minus infinity.

For example:

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25;      // Yields: 1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25;     // Yields: -1.5
```

AP_TRN_ZERO

The AP_TRN_ZERO quantization mode indicates that the value should be rounded to the nearest representable value.

* For positive values the rounding is same as mode AP_TRN.
* For negative values, round towards zero.

For example:

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25;   // Yields: -1.0
```

Overflow modes

Saturation	AP_SAT
Saturation to zero	AP_SAT_ZERO
Symmetrical saturation	AP_SAT_SYM
Wrap-around	AP_WRAP
Sign magnitude wrap-around	AP_WRAP_SM

AP_SAT

The AP_SAT overflow mode indicates the value should be saturated to the maximum value in case of overflow, or to the negative maximum value in case of negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0;     // Yields: 15.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0;      // Yields: 7.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0;    // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0;     // Yields: -8.0
```

AP_SAT_ZERO

The AP_SAT_ZERO overflow mode indicates the value should be forced to in case of overflow, or negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0;   // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0;    // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0;  // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0;   // Yields: 0.0
```

AP_SAT_SYM

The AP_SAT_SYM overflow mode indicates the value should be saturated to the maximum value in case of overflow, or to the minimum value (negative maximum for signed ap_fixed types and zero for unsigned ap_ufixed types) in case of negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0;    // Yields: 15.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0;     // Yields: 7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0;   // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0;    // Yields: -8.0
```

AP_WRAP

The AP_WRAP overflow mode indicates that the value should be wrapped around in case of overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0;       // Yields: 3.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0;        // Yields: -1.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0;      // Yields: 13.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0;       // Yields: -3.0
```

If the value of N is set to zero (the default overflow mode):

- Any MSB bits outside the range are deleted.
- For unsigned numbers: after the maximum it wraps around to zero
- For signed numbers: after the maximum, it wraps to the minimum values.

If N>0:

- When N > 0, N MSB bits are saturated or set to 1.
- The sign bit is retained, so positive numbers remain positive and negative numbers remain negative.
- The bits that are not saturated are copied starting from the LSB side.

AP_WRAP_SM

The AP_WRAP_SM overflow mode indicates that the value should be sign-magnitude wrapped around.

For example:

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0;     // Yields: -4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0;    // Yields: 2.0
```

If the value of N is set to zero (the default overflow mode):

- This mode uses sign magnitude wrapping.
- Sign bit set to the value of the least significant deleted bit.
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted.
- IF MSBs are same, the other bits are copied over.
  - Step 1: First delete redundant MSBs.
  - Step 2: The new sign bit is the least significant bit of the deleted bits. 0 in this case.
  - Step 3: Compare the new sign bit with the sign of the new value.
- If different, invert all the numbers. They are different in this case.

If N>0:

- Uses sign magnitude saturation
- N MSBs will be saturated to 1.
- Behaves similar to case where N = 0, except that positive numbers stay positive and negative numbers stay negative.

Compiling ap_[u]fixed<> Types

In order to use the `ap_[u]fixed<>` classes one must include the "ap_fixed.h" header file in all source files which reference `ap_[u]fixed<> variables`.

When compiling software models that use these classes, it may be necessary to specify the location of the Vivado HLS header files, for example by adding the "`-I/<HLS_HOME>/include`" option for `g++` compilation.

Also note that best performance will be observed for software models when compiled with `g++ -O3` option.

Declaring/Defining ap_[u]fixed<> Variables

There are separate signed and unsigned classes: `ap_fixed<W,I>` and `ap_ufixed<W,I>` respectively.

As usual, user defined types may be created with the C/C++ '`typedef`' statement as shown among the following examples:

```
include "ap_fixed.h"                    // use ap_[u]fixed<> types

typedef ap_ufixed<128,32> uint128_t;    // 128-bit user defined type,
                                        //  32 integer bits
```

Initialization and Assignment from Constants (Literals)

Any `ap_[u]fixed` variable may be initialized with normal floating point constants of the usual C/C++ width (32 bits for type float, 64 bits for type double). That is, typically, a floating point value that is single precision type or in the form of double precision.

Such floating point constants will be interpreted and translated into the full width of the arbitrary precision fixed-point variable depending on the sign of the value.

For example:

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
```

Support for console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, features are provided to support printing values which require more than 64-bits to represent.

The easiest way to output any value stored in an `ap_[u]int` variable is to use the C++ standard output stream, `std::cout (#include <iostream> or <iostream.h>)`. The stream insertion operator, "`<<`", is overloaded to correctly output the full range of values possible for any given `ap_[u]int` variable. The stream manipulators "`dec`", "`hex`" and "`oct`" are also supported, allowing formatting of the value as decimal, hexadecimal or octal respectively.

Example using "`cout`" to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_uint<72> Val("10fedcba9876543210");

cout << Val << endl;              // Yields: "313512663723845890576"
cout << hex << val << endl;       // Yields: "10fedcba9876543210"
cout << oct << val << endl;       // Yields: "41773345651416625031020"
```

It is also possible to use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits, by first converting the value to a C++ `std::string`, then to a C character string. The `ap_[u]int` classes provide a method, `to_string()` to do the first conversion and the `std::string` class provides the `c_str()` method to convert to a null-terminated character string.

The `ap[u]int::to_string()` method may be passed an optional argument specifying the radix of the numerical format desired. The valid radix argument values are 2, 8, 10 and 16 for binary, octal, decimal and hexadecimal respectively; the default radix value is 16.

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean and the default value is `false`, causing the non-decimal formats to be printed as unsigned values.

Examples for using `printf` to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str();           // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str();         // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str();          // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str();   // => "-7F0123456789ABCDF0"
```

Expressions Involving ap_[u]fixed<> types

Arbitrary precision fixed-point values can participate in expressions that use any operators supported by C/C++. Once an arbitrary precision fixed-point type or variable is defined, their usage is the same as for any floating point type or variable in the C/C++ languages. However, there are a few caveats:

Zero and Sign Extensions

Remember that all values of smaller bit-width will be zero or sign extended depending on the sign of the source value. You may need to insert casts to obtain alternative signs when assigning smaller bit-width to larger.

Truncations

When you assign an arbitrary precision fixed-point of larger bit-width than the destination variable, truncation will occur.

Class Operators & Methods

In general, any valid operation that may be done on a native C/C++ integer data type, is supported, via operator overloading, for `ap_[u]fixed` types. In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Addition

```
ap_[u]fixed::RType ap_[u]fixed::operator + (ap_[u]fixed op)
```

This operator function adds this arbitrary precision fixed-point with a given operand `op` to produce a result.

TIP: *The operands can be ap_[u]fixed, ap_[u]int, or C/C++ integer types. The result type ap_[u]fixed::RType depends on the type information of the two operands.*

For example:

```
ap_fixed<76, 63> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 + Val2;   //Yields 6722.480957
```

Since `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one, in order to be able to store all possible result values.

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - (ap_[u]fixed op)
```

This operator function subtracts this arbitrary precision fixed-point with a given operand `op` to produce a result.

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

For example:

```
ap_fixed<76, 63> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 - Val1; // Yields 6720.23057
```

Since `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one, in order to be able to store all possible result values.

Multiplication

```
ap_[u]fixed::RType ap_[u]fixed::operator * (ap_[u]fixed op)
```

This operator function multiplies this arbitrary precision fixed-point with a given operand `op` to produce a result.

For example:

```
ap_fixed<80, 64> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 * Val2;   // Yields 7561.525452
```

We have the multiplication of Val1 and Val2. The result type has sum of their integer part bit-width and their fraction part bit width.

Division

```
ap_[u]fixed::RType ap_[u]fixed::operator / (ap_[u]fixed op)
```

This operator function divides this arbitrary precision fixed-point by a given operand op to produce a result.

For example:

```
ap_fixed<84, 66> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 / Val1;            // Yields 5974.538628
```

We have the division of Val1 and Val2. In order to preserve enough precision:

- The integer bit-width of the result type is sum of the integer = bit-width of Val1 and the fraction bit-width of Val2.
- The fraction bit-width of the result type is sum of the fraction bit-width of Val1 and the whole bit-width of Val2.

Bitwise Logical Operators

Bitwise OR

```
ap_[u]fixed::RType ap_[u]fixed::operator | (ap_[u]fixed op)
```

This operator function applies or bitwise operation on this arbitrary precision fixed-point and a given operand op to produce a result.

For example:

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 | Val2;            // Yields 6271.480957
```

Bitwise AND

```
ap_[u]fixed::RType ap_[u]fixed::operator & (ap_[u]fixed op)
```

This operator function applies and bitwise operation on this arbitrary precision fixed-point and a given operand op to produce a result.

For example:

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 & Val2;              // Yields 1.00000
```

Bitwise XOR

```
ap_[u]fixed::RType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

This operator function applies `xor` bitwise operation on this arbitrary precision fixed-point and a given operand `op` to produce a result.

For example:

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 ^ Val2;          // Yields 6720.480957
```

Increment and Decrement Operators

Pre-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ ()
```

This operator function prefix increases this arbitrary precision fixed-point variable by `1` to produce a result.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ++Val1;               // Yields 6.125000
```

Post-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ (int)
```

This operator function postfix increases this arbitrary precision fixed-point variable by `1`, returns the original val of this arbitrary precision fixed-point.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1++;               // Yields 5.125000
```

Pre-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- ()
```

This operator function prefix decreases this arbitrary precision fixed-point variable by `1` to produce a result.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = --Val1;              // Yields 4.125000
```

Post-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- (int)
```

This operator function postfix decreases this arbitrary precision fixed-point variable by `1`, returns the original val of this arbitrary precision fixed-point.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1--;              // Yields 5.125000
```

Unary Operators

Addition

```
ap_[u]fixed ap_[u]fixed::operator + ()
```

This operator function returns self copy of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1;               // Yields 5.125000
```

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - ()
```

This operator function returns negative value of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;
```

```
Result = -Val1;                // Yields -5.125000
```

Equality Zero

```
bool ap_[u]fixed::operator ! ()
```

This operator function compares this arbitrary precision fixed-point variable with `0`, and returns the result.

For example:

```
bool  Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1;                // Yields false
```

Bitwise Inverse

```
ap_[u]fixed::RType ap_[u]fixed::operator ~ ()
```

This operator function returns bitwise complement of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ~Val1;                // Yields -5.25
```

Shift Operators

Unsigned Shift left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_uint<_W2> op)
```

This operator function shifts left by a given integer operand, and returns the result. The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift left operation is the same width as type being shifted.

Note: Shift currently cannot support overflow or quantization modes.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val << sh;            // Yields -10.5
```

The bit-width of the result is (`W = 25`, `I = 15`), however, since the shift left operation result type is same as the type of `Val`, the high order two bits of `Val` are shifted out, and the result is -10.5.

If a result of 21.5 is required, `Val` must be cast to `ap_fixed<10, 7>` first: e.g., `ap_ufixed<10, 7>(Val)`.

Signed Shift Left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_int<_W2> op)
```

This operator shifts left by a given integer operand, and returns the result. The shift direction depends on whether operand is positive or negative.

- If the operand is positive, a shift right performed.
- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift right operation is the same width as type being shifted.

For example:

```
ap_fixed<25, 15,  false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val << sh;           // Shift left, yields -10.25

Sh = -2;
Result = Val << sh;           // Shift right, yields 1.25
```

Unsigned Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_uint<_W2> op)
```

This operator function shifts right by a given integer operand, and returns the result. The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift right operation is the same width as type being shifted.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val >> sh;           // Yields 1.25
```

If it is required to preserve all significant bits, extend fraction part bit-width of the `Val` first, for example `ap_fixed<10, 5>(Val)`.

Signed Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_int<_W2> op)
```

This operator shifts right by a given integer operand, and returns the result. The shift direction depends on whether operand is positive or negative.

- If the operand is positive, a shift right performed.
- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift right operation is the same width as type being shifted. For example:

```
ap_fixed<25, 15,  false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val >> sh;           // Shift right, yields 1.25

Sh = -2;
Result = Val >> sh;           // Shift left,  yields -10.5

1.25
```

Relational Operators

Equality

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

This operator compares the arbitrary precision fixed-point variable with a given operand, and returns true if they are equal and false if they are not equal.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types. For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 == Val2;        // Yields  true
Result = Val1 == Val3;        // Yields  false
```

Non-Equality

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

This operator compares this arbitrary precision fixed-point variable with a given operand, and returns `true` if they are not equal and `false` if they are equal.

The type of operand op can be ap_[u]fixed, ap_int or C/C++ integer types. For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 != Val2;        // Yields false
Result = Val1 != Val3;        // Yields true
```

Greater-than-or-equal

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and returns `true` if they are equal or if the variable is greater than the operator and `false` otherwise.

The type of operand `op` can be `ap_[u]fixed, ap_int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 >= Val2;        // Yields true
Result = Val1 >= Val3;        // Yields false
```

Less-than-or-equal

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is equal to or less than the operand and `false` if not.

The type of operand op can be `ap_[u]fixed`, `ap_int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 <= Val2;        // Yields true
Result = Val1 <= Val3;        // Yields true
```

Greater-than

```
bool ap_[u]fixed::operator > (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is greater than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 > Val2;          // Yields false
Result = Val1 > Val3;          // Yields false
```

Less-than

```
bool ap_[u]fixed::operator < (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is less than the operand and `false` if not.

The type of operand op can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types. For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 < Val2;          // Yields false
Result = Val1 < Val3;          // Yields true
```

Bit Operator

Bit-Select-and-Set

```
af_bit_ref ap_[u]fixed::operator [] (int bit)
```

This operator selects one bit from an arbitrary precision fixed-point value and returns it.

The returned value is a reference value, which can be used to set or clear the corresponding bit in the `ap_[u]fixed` variable. The bit argument must be an integer value and it specifies the index of the bit to select. The least significant bit has index `0`. The highest permissible index is one less than the bit-width of this `ap_[u]fixed` variable.

The result type is `af_bit_ref` with a value of either `0` or `1`. For example:

```
ap_int<8, 5> Value = 1.375;

Value[3];                      // Yields  1
```

```
Value[4];                       // Yields  0

Value[2] = 1;                   // Yields 1.875
Value[3] = 0;                   // Yields 0.875
```

Bit Range

```
af_range_ref af_(u)fixed::range (unsigned Hi, unsigned Lo)
af_range_ref af_(u)fixed::operator [] (unsigned Hi, unsigned Lo)
```

This operation is similar to bit-select operator [] except that it operates on a range of bits instead of a single bit.

It selects a group of bits from the arbitrary precision fixed point variable. The `Hi` argument provides the upper range of bits to be selected. The `Lo` argument provides the lowest bit to be selected. If `Lo` is larger than `Hi` the bits selected will be returned in the reverse order.

The return type `af_range_ref` represents a reference in the range of the `ap_[u]fixed` variable specified by `Hi` and `Lo`. For example:

```
ap_uint<4> Result = 0;
ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(3, 0);             // Yields: 0x5
Value(3, 0) = Repl(3, 0);               // Yields: -1.5

// when Lo > Hi, return the reverse bits string
Result = Value.range(0, 3);             // Yields: 0xA
```

Range Select

```
af_range_ref af_(u)fixed::range ()
af_range_ref af_(u)fixed::operator []
```

This operation is the special case of the range select operator `[]`. It selects all bits from this arbitrary precision fixed point value in the normal order.

The return type af_range_ref represents a reference to the range specified by Hi = W - 1 and Lo = 0. For example:

```
ap_uint<4> Result = 0;

ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range();             // Yields: 0x5
Value() = Repl(3, 0);               // Yields: -1.5
```

Length

```
int ap_[u]fixed::length ()
```

This function returns an integer value that provides the number of bits in an arbitrary precision fixed-point value. It can be used with a type or a value. For example:

```
ap_ufixed<128, 64> My128APFixed;

int bitwidth = My128APFixed.length();   // Yields 128
```

Explicit Conversion to Methods

Fixed-toDouble

```
double ap_[u]fixed::to_double ()
```

This member function returns this fixed-point value in form of IEEE double precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
double Result;

Result = MyAPFixed.to_double();         // Yields 333.789
```

Fixed-to-ap_int

```
ap_int ap_[u]fixed::to_ap_int ()
```

This member function explicitly converts this fixed-point value to `ap_int` that captures all integer bits (fraction bits are truncated). For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
ap_uint<77> Result;

Result = MyAPFixed.to_ap_int();      //Yields 333
```

Fixed-to-integer

```
int ap_[u]fixed::to_int ()
unsigned ap_[u]fixed::to_uint ()
ap_slong ap_[u]fixed::to_int64 ()
ap_ulong ap_[u]fixed::to_uint64 ()
```

This member function explicitly converts this fixed-point value to C built-in integer types. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
unsigned int  Result;

Result = MyAPFixed.to_uint();        //Yields 333

unsigned long long Result;
Result = MyAPFixed.to_uint64();      //Yields 333
```

XILINX®

Chapter 5

High-Level Synthesis Command Reference Guide

Using High-Level Synthesis Commands

Before using any High-Level Synthesis commands in an active design project it is worthwhile to understand and appreciate a few basic High-Level Synthesis concepts.

1. High-Level Synthesis stores design in a project based structure.
2. High-Level Synthesis optimizations are specified on regions, or locations, within the code.
3. High-Level Synthesis optimizations can be specified as Tcl commands or pragmas in the source code (and in addition to editing text files, both options can be performed from within the High-Level Synthesis GUI).

Each of these concepts is fully explained in the remaining sections of this chapter.

Managing Projects

High-Level Synthesis uses a project based database to manage the synthesis and verification processes and to store the results. Every design must be represented in a project which may itself contain multiple solutions.

The source code and testbench are stored in the project. Solutions can be used to specify a different target technology (different FPGA families and devices), apply different directives and create different implementations of the same source code, with a view to selecting the most optimum implementation.

High-Level Synthesis projects and solutions are directly reflected in the directory structure used by High-Level Synthesis. Figure 1-1 shows an example High-Level Synthesis directory after results have been generated.

In the example in Figure 1-1:

- The project, as shown on the top line, is called `project.prj` and all project data is stored in a project directory of the same name.

- The project contains source code and testbench files.
- There are 2 solutions in this project (`solution1` and `solution2`).
- Currently **`solution1`** (highlighted in bold) is the active solution.
- Simulation results have been generated, as shown by the `sim` directory.
- Synthesis results have been generated, as shown by the `syn` directory.
  - The `syn` directory shows RTL output has been created in verilog, VHDL and SystemC.
  - The `syn` directory shows reports have been generated.

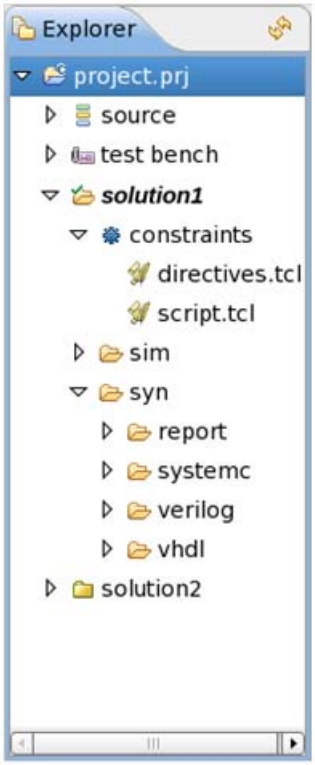

Figure 5-1: **Project and Solution Structure**

When using High-Level Synthesis commands, it is important to understand that some commands can only be used inside an active project or solution. In general, projects are used for the same set of source code and solutions are used to create different implementations of that source code.

Note: Opening a new project automatically closes the existing project and opening a new solution automatically closes the existing solution.

High-Level Synthesis Optimization Locations

Within High-Level Synthesis optimizations can be specified on functions, loops, regions, arrays and interface parameters. This section explains how optimizations are applied and the locations they affect.

Optimizations are specified in one of three ways:

1. Using the directives tab in the GUI.
2. Tcl commands can be used at the interactive prompt, or with a batch file, to specify directives, provided the object can be uniquely identified (loops and regions require a label).
3. Insert pragmas directives directly into the source code.

The optimizations specified by all of these techniques are applied to the specified location (scope) within the source code.

The following example shows the outline of some source code.

```
int foo_sub_A (int mem_1[64],..) {
     for_A: for (int n = 0; n < 3; ++n) {
               ...
     }
     ...
}
int foo_sub_B (int mem_1[64], int i) {
     for_B:for (int n = 0; n < 4; ++n) {
               ...
     }
     ...
}
void foo_top (int mem_1[64], int mem_2[64]) {
     ...
     for_top: for (int i = 0; i < 64; ++i) {
            my_label: {
                    ...
            }
     }
}
```

- Figure 1-2 shows how this code is represented in the GUI directives tab. The directives tab can be viewed by selecting the source code within the Project Explorer, then selecting the directives tab in the Auxiliary Pane (right hand side of the GUI).

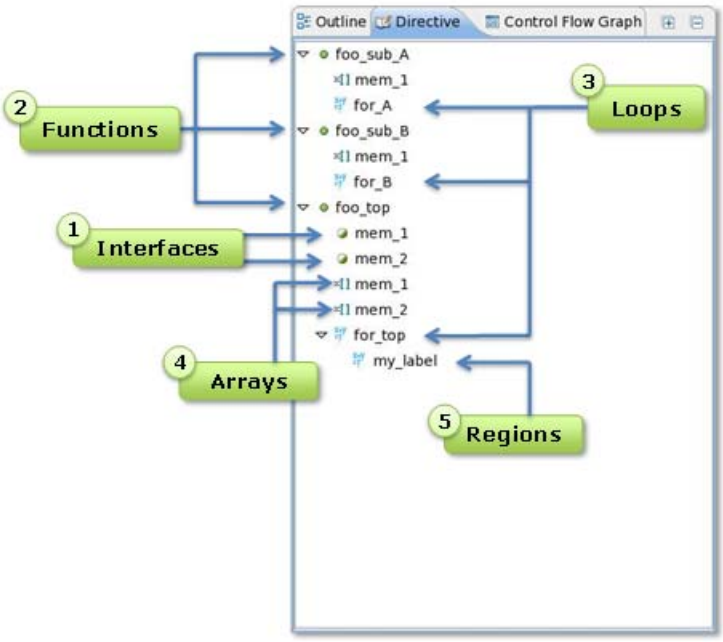

Figure 5-2: **GUI Directives Objects**

1. Interfaces - Directives applied to interfaces are applied to that interface object (function parameter, global or return value) and nothing else.
2. Functions - Directives applied to functions are performed on all objects within the scope of the function. The effect of any directive stops at the next level of function hierarchy except in the case of PIPELINE which recursively flattens and unrolls everything, or if the directive supports a recursive option and it is applied.
   a. In the example in Figure 1-2, a directive applied on `foo_top` does not affect the operations in `foo_sub_A` or `foo_sub_B` (other than the exceptions stated above).
3. Loops - Directives applied on loops impacts all objects within the scope of the loop.
   a. If a MERGE directive is applied to a loop, the directive applies to any sub-loops but not to the loop itself. The loop is not merged with siblings at the same level of hierarchy but any sub-loops are merged.
   b. A PIPELINE directive also applies to objects within the loop: they are pipelined, which is essentially the same as pipelining the loop itself.
4. Arrays - Directives can be applied to arrays directly, in which case they only apply to the array itself, or they can be applied to functions, loops or regions which contain multiple arrays, in which case the directive applies to all arrays enclosed.
5. Regions - Regions of code can be created by inserting a pair of braces: {the code between is a region}. Any directive applied to a labeled region applies to the objects within that region.

Note: To apply directives using Tcl commands, loops and regions must have a named label as shown in the above example (loop label `for_top` and region label `my_label`)

Other than interfaces and arrays, the other objects (function, loops and regions) represent "areas" of the code which can cover multiple statements. For this reason, the command pages within this manual lists the target as "location". Keep in mind when a directive is applied to a location it is applied to all objects within the location (unless otherwise explicitly specified).

Commands and Pragmas

High-Level Synthesis optimizations are primarily based on locations within the code, as discussed in the previous section. This model for specifying optimizations supports the use of pragmas in the code.

In this example, a pragma is inserted into the code to unroll a for-loop. After the pragma, keyword `AP` is always specified and then the directive:

```
for (int i = 0; i < 64; ++i) {
#pragma AP UNROLL
  ...
}
```

Any directives entered in the C code are shared by every implementation which uses the C specification. This could be a desired feature, to have the C specification contain all the optimization directives, however in many cases designers may wish to separate the pragmas from the source to allow different solutions to be created (when pragmas are used, every solution which uses the source will have the same optimizations performed).

In the above example, the command `set_directive_loop_unroll` could have been used at the command line interface provided the loop had a named label or the unroll directive could have been added to the source code using the GUI directives tab.

The following can be viewed using the `help` command within High-Level Synthesis and lists the associated pragma for each optimization directive:

```
>autopilot help
...
set_directive_allocation           - Directive ALLOCATION
set_directive_array_map            - Directive ARRAY_MAP
set_directive_array_partition      - Directive ARRAY_PARTITION
set_directive_array_reshape        - Directive ARRAY_RESHAPE
set_directive_array_stream         - Directive ARRAY_STREAM
set_directive_dataflow             - Directive DATAFLOW
set_directive_dependence           - Directive DEPENDENCE
set_directive_expression_balance   - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline               - Directive INLINE
set_directive_interface            - Directive INTERFACE
set_directive_latency              - Directive LATENCY
set_directive_loop_flatten         - Directive LOOP_FLATTEN
set_directive_loop_merge           - Directive LOOP_MERGE
set_directive_loop_tripcount       - Directive LOOP_TRIPCOUNT
set_directive_occurrence           - Directive OCCURRENCE
set_directive_pipeline             - Directive PIPELINE
```

```
set_directive_platform            - Directive PLATFORM
set_directive_power               - Directive POWER
set_directive_protocol            - Directive PROTOCOL
set_directive_resource            - Directive RESOURCE
set_directive_top                 - Directive TOP
set_directive_unroll              - Directive UNROLL
...
```

The easiest way to determine what can be optimized and what the command or pragma is:

1. Open the code in the GUI.
2. Select the **Directive** tab, as shown in Figure 1-1, which shows all objects which can be optimized.
3. With the object selected, right-click with the mouse to specify a directive.
   a. Select the **Destination** as **Into Directive File** to see the command in `constraints/directive.tcl`.

   OR

   b. Select the **Destination** as **Into Source File** to see the pragma inserted directly into the code.

High-Level Synthesis Commands

add_file

Syntax

```
add_file [OPTIONS] <src_files>
```

Description

The `add_file` command adds design source files to the current project. The current directory is automatically searched for any header files included in the design source. To use header files stored in directories other than the current directory, use the **`-cflags`** option to add those directories to the search path.

<src_files> - A list of source files with the description of the design.

Options

`-tb` - Specify any files used as part of the design testbench. These files are not synthesized but used when post-synthesis verification is executed by the `cosim_design` command. No design files can be included in the list of source files when this option is used: use a separate `add_file` command to add design files and testbench files.

`-cflags` *<string>* - A string with any desired GCC compilation options.

`-type` (**`c`** | **`sc`**) - Specify if the source files are a C/C++ (c) or a SystemC (sc). The default is C/C++ source files.

Pragma

There is no pragma equivalent of the `add_file` command.

Examples

This example adds three design files to the project.

```
add_file a.cpp
add_file b.cpp
add_file c.cpp
```

Multiple files can be added with a single command line.

```
add_file "a.cpp b.cpp c.cpp"
```

The following example adds a SystemC file with compiler flags to enable macro USE_RANDOM and specifies an additional search path, sub-directory `./lib_functions`, for header files.

```
add_file -type sc top.cpp -cflags "-DUSE_RANDOM -I./lib_functions"
```

The **`-tb`** option is used to add testbench files to the project. This example adds multiple files with a single command, including the testbench `a_test.cpp` and all data files read by the testbench, `input_stimuli.dat` and `out.gold.dat`.

```
add_file -tb "a_test.cpp input_stimuli.dat out.gold.dat"
```

If the testbench data files in the previous example are stored in a separate directory, for example `test_data`, the directory can be added to the project in place of the individual data files.

```
add_file -tb a_test.cpp
add_file -tb test_data
```

autoimpl

Syntax

`autoimpl` [*OPTIONS*]

Description

Automatically creates and executes the scripts to create a gate-level implementation of the RTL.

You can specify that a specific user-provided script to be used for the implementation in place of the default High-Level Synthesis generated script.

The implementation scripts are created in sub-directory `impl/<rtl>` of the active solution. By default, `autoimpl` automatically executes the scripts in this same directory, to produce a gate level implementation. The **`-setup`** option can be used to create the scripts only: logic synthesis is not invoked.

The **`-export`** and **`-custom_ports`** options can be used to create a pcore implementation for use within the EDK environment. The option **`-xil_coregen`** will call the Core Generator tool to create and instantiate optimized components (such as BRAM, Floating-Point blocks) prior to logic synthesis.

Options

-export - This option creates an additional implementation of the design, with the appropriate wrappers and external adapters, which can be imported into EDK as a core model (can be directly copied to a project pcores directory).

-par - This option can only be used with the **-tool** option when specifying Synopsys Synplify products for RTL synthesis. When this option is specified the place and route implementation will be executed using the ISE Design Suite (otherwise place and route is not performed).

-rtl (**verilog**|**vhdl**) - Determines which HDL is used for the RTL implementation. The default is **verilog**.

-setup - When this option is specified all the implementation files will be created in the impl/<*rtl*> directory of the active solution but the implementation will not be executed.

-tool - Specify which RTL synthesis tool is used to create a gate-level implementation. The options are:

- **ise** - Xilinx ISE Design Suite (default)
- **synplify** - Synopsys Synplify

The **-tool** option can also be used to pass a string with additional tool options. This option is only available using the command line and is not available in the GUI. For example, to specify that Synplify Professional use a specific license, the following can be used:

```
autoimpl -tool "synplify_pro -licensetype synplifypro_xilinx"
-xil_coregen
```

The string option can also be used to specify the exact executable. For example, synplify_pro, synplify_pro_dp or synplify_premier:

```
autoimpl - tool "synplify_premier"
```

This option uses the CORE Generator tool flow to implement optimized netlists for components in the RTL (such as BRAM/FP).

Pragma

There is no pragma equivalent of the `autoimpl` command.

Examples

This example creates all the scripts for implementation tool, but does not start implementation.

```
autoimpl -setup
```

The following example implements the design using Synplify (with the default Verilog RTL):

```
autoimpl -tool synplify
```

This example creates the scripts for implementing the VHDL using the ISE Design Suite, but does not execute the ISE Design Suite, calls the CORE Generator tool to create optimized modules (such as memories, FIFOs) which are instantiated in the RTL prior to synthesis and creates a pcore directory for use with EDK.

```
autoimpl -rtl vhdl -setup -xil_coregen -export
```

cosim_design

Syntax

cosim_design [*OPTIONS*]

Description

Executes post-synthesis co-simulation of the synthesized RTL with the original C-based testbench. The files for the testbench are specified with the `add_file` **-tb** command and option. The simulation is run in sub-directory `sim/<`*HDL*`>` of the active solution, where <*HDL*> is selected by the **-rtl** option.

For a design to be verified with `cosim_design`, the design must use interface mode **ap_ctrl_hs** and each output port must use one of the following interface modes which identify, with a write valid signal, when an output is written: **ap_vld**, **ap_ovld**, **ap_hs**, **ap_memory**, **ap_fifo** or **ap_bus**.

Options

-O - Enables optimize compilation of the C testbench and RTL wrapper. Without optimization, `cosim_design` will compile the testbench as quickly as possible. Enable optimization to improve the run time performance, if possible, at the expense of compilation time.

Note: Although the resulting executable may potentially run much faster the run time improvements are design dependent. Optimizing for run time may require large amounts of memory for large functions.

-argv <*string*> - Option to specify the argument list for the behavioral testbench. The <string> will be passed onto the main C function.

-coverage - This option enables the coverage feature during simulation with the VCS simulator.

-ignore_init <*integer*> - Disables comparison checking for the first <integer> number of clock cycles. This is useful when it is know the RTL will initially start with unknown ('hX) values.

-ldflags <*string*> - Provides for the specification of options that need to be passed to the linker for co-simulation. This is typically used to pass on include path information or library information for the C test bench.

-mflags <*string*> - Provides for the specification of options that need to be passed to the compiler for SystemC simulation. This is typically used to speed up compilation.

-rtl (**systemc**|**vhdl**|**verilog**) - Selects which RTL is to be used for verification with the C testbench. For Verilog and VHDL a simulator must be specified with the **-tool** option. The default is **systemc**.

-setup - When this option is specified all the simulation files will be created in the `sim/<HDL>` directory of the active solution but the simulation will not be executed.

-tool (**vcs**|**modelsim** |**riviera**) - Option to select the simulator to be used to co-simulate the RTL with the C testbench. No tool needs to be specified for SystemC co-simulation: High-Level Synthesis will use its included SystemC kernel.

-trace_level (**none**|**all**) - Determines the level of VCD output which is performed. Option **all** results in all ports and signals being saved to the VCD file. The VCD file is saved in the `sim/<HDL>` directory of the current solution when the simulation executes. The default is **none**.

Pragma

There is no pragma equivalent of the `cosim_design` command.

Examples

Perform verification using the SystemC RTL.

```
cosim_design
```

Use the VCS simulator to verify the Verilog RTL, enable the VCD coverage feature and save all signals in VCD format.

```
cosim_design -tool VCS -rtl verilog -coverage -trace_level all
```

In this example, the VHDL RTL is verified using ModelSim and values 5 and 1 are passed to the testbench function and used in the RTL verification.

```
cosim_design -tool modelsim -rtl vhdl -argv "5 1"
```

This final example creates an optimized simulation model for the SystemC RTL but does not execute the simulation. To run the simulation, execute `run.sh` in the `sim/systemc` directory of the active solution.

```
cosim_design -O -setup
```

autosyn

Syntax

```
autosyn
```

Description

The `autosyn` command synthesizes the High-Level Synthesis database for the active solution. The command can only be executed in the context of an active solution. The elaborated design in the database is scheduled and mapped onto RTL, based on any constraints that are set.

Pragma

There is no pragma equivalent of the `autosyn` command.

Examples

Run High-Level Synthesis synthesis on the top-level design.

```
autosyn
```

close_project

Syntax

```
close_project
```

Description

The `close_project` command closes the current project, so this project is no longer active in the High-Level Synthesis session. The command prevents the designer from entering any project or solution specific commands, but is not really required since opening or creating a new project will automatically close the current active project.

Pragma

There is no pragma equivalent of the `close_project` command.

Examples

Close the current project. All results are automatically saved.

```
close_project
```

close_solution

Syntax

```
close_solution
```

Description

The `close_solution` command closes the current solution, so this solution is no longer active in the High-Level Synthesis session. The command prevents the designer from entering any solution specific commands, but is not really required since opening or creating a new solution will automatically close the current active solution.

Pragma

There is no pragma equivalent of the `close_solution` command.

Examples

Close the current solution. All results are automatically saved.

```
close_solution
```

config_array_partition

Syntax

```
config_array_partition [OPTIONS]
```

Description

This command allows the default behavior for array partitioning to be specified.

OPTIONS

-auto_partition_threshold <*int*> - Sets the threshold for automatically partitioning arrays (including those without constant indexing). Arrays which have fewer elements than the specified threshold limit will be automatically partitioned into individual elements, unless interface/core specification is applied on the array. The default is 4.

-auto_promotion_threshold <*int*> - Sets the threshold for automatically partitioning arrays with constant-indexing. Arrays which have fewer elements than the specified threshold limit and have constant-indexing (the indexing is not variable) will be automatically partitioned into individual elements. The default is 64.

-exclude_extern_globals - Excludes external global arrays from throughput driven auto-partitioning. By default, external global arrays are partitioned when option **-throughput_driven** is selected. This option has no effect unless option **-throughput_driven** is selected.

-include_ports - Enables auto-partitioning of IO arrays. This has the effect of reducing an array IO port into multiple ports, each port the size of the individual array elements.

-scalarize_all - This option partitions all arrays in the design into their individual elements.

-throughput_driven - Enables auto-partitioning of arrays based on the throughput. High-Level Synthesis will automatically determine if partitioning the array into individual elements will allow it to meet any specified throughput requirements.

Pragma

There is no pragma equivalent of the `config_array_partition` command.

Examples

In this example, all arrays in the design with less than 12 elements, but not global arrays, are automatically partitioned into individual elements.

```
config_array_partition auto_partition_threshold 12 -exclude_extern_globals
```

This example instructs High-Level Synthesis to automatically determine which arrays to partition, including arrays on the function interface, to improve throughput.

```
config_array_partition -throughput_driven -include_ports
```

Partition all arrays in the design, including global arrays, into individual elements.

```
config_array_partition -scalarize_all
```

config_bind

Syntax

```
config_bind [OPTIONS]
```

Description

This command allows the default options for micro-architecture binding to be set. Binding is the process where operators, such as addition, multiplication, and shift are mapped to specific RTL implementations; for example a mult operation implemented as a combinational or pipelined RTL multiplier.

Options

-effort (**low**|**medium**|**high**) - The optimizing effort level controls the trade off between run-time and optimization. The default is **medium** effort.

A **low** effort optimization will improve the run time and may be useful for cases where little optimization is possible, for example when all if-else statements have mutually exclusive operators in each branch and no operator sharing can be achieved.

A **high** effort optimization will result in increased run time but will typically provide better quality of results.

-min_op <*string*> - This option tells High-Level Synthesis to minimize the number of instances of a particular operator. If there are multiple such operators in the code, this will result in them being shared onto the fewest number of RTL resources (cores).

The following operators can be specified as arguments to this command option:

- **add** - Addition
- **sub** - Subtraction
- **mul** - Multiplication
- **icmp** - Integer Compare
- **sdiv** - Signed Division
- **udiv** - Unsigned Division
- **srem** - Signed Remainder
- **urem** - Unsigned Remainder

- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **shl** - Shift-Left

Pragma

There is no pragma equivalent of the config_bind command.

Examples

This example tells High-Level Synthesis to spend more effort in the binding process, try more options for implementing the operators, and try to produce a design with better resource usage.

```
config_bind -effort high
```

In this example, the number of multiplication operators is minimized, resulting in RTL with the fewest number of multipliers.

```
config_bind -min_op mul
```

config_dataflow

Syntax

config_dataflow [*OPTIONS*]

Description

This command specifies the default behavior of dataflow pipelining (implemented by the set_directive_dataflow command). This configuration command allows you to specify the default channel memory type and depth.

Options

-default_channel (**fifo**|**pingpong**) - By default a RAM memory, configured in **pingpong** fashion, is used to buffer the data between functions or loops when dataflow pipelining is used. When streaming data is used (where the data is always read and written in consecutive order), a FIFO memory will be more efficient and can be selected as the default memory type.

Note: Arrays must be set to streaming using the set_directive_array_stream command in order to perform FIFO accesses.

-fifo_depth <*integer*> - An integer value specifying the default depth of the FIFOs. This option has no effect when pingpong memories are used. If not specified the FIFOs used in the channel will be set to the size of the largest producer or consumer (whichever is largest). In some cases this may be too conservative and introduce FIFOs which are larger than they need to be. This option can be used when you know the FIFOs are larger than they are required to be. Be careful when using this option as incorrect use may result in a design which fails to operate correctly.

Pragma

There is no pragma equivalent of the `config_dataflow` command.

Examples

Change the default channel from ping-pong memories to a FIFO.

```
config_dataflow -default_channel
```

Change the default channel from ping-pong memories to a FIFO with a depth of 6.

Note: If the design implementation requires a FIFO with greater than 6 elements, this setting will result in which fails RTL verification: this option is a user over-ride and care should be taken when using it.

```
config_dataflow -default_channel fifo -fifo_depth 6
```

config_interface

Syntax

```
config_interface [OPTIONS]
```

Description

The `config_interface` command specifies the default interface used to implement the RTL port of each function argument during interface synthesis. The function arguments can be pass-by-value variables, pointers, arrays and pass-by-reference variables (as permitted by the input language).

In addition, the `config_interface` command can be used to specify the default interface for function level control (such as start, done) and to expose any global variables used by the function as ports on the RTL design.

The default interface is used if none is explicitly specified for the function argument or if an incompatible interface type is specified. A complete list of all interface types is provided

below and a detailed explanation of each interface is provided in the *High-Level Synthesis User Guide (UG867)*.

Interface Types

ap_none - This interface provides no additional handshake or synchronization ports and can be applied to any function argument type except arrays. This is the default interface type for all read-only arguments (input ports), except array arguments.

ap_ack - Can be specified on any function argument except arrays and provides an additional acknowledge port to indicate input data has been read by this RTL block or confirm output data has been read by a downstream RTL block.

ap_vld - Can be specified on any function argument except arrays and provides an additional data-valid port to indicate when input data is valid and can be read or when output data is valid.

ap_ovld - Identical to the **ap_vld** interface except that it only applies to write-only arguments (RTL output ports). This is the default interface type for all write-only arguments, except array arguments.

ap_hs - Implements each argument with an RTL port supported by a full two-way acknowledge and valid handshake. This can be specified for any function argument except arrays.

ap_fifo - An **ap_fifo** interface can be specified for pointer, array or pass-by-reference arguments. An **ap_fifo** interface implements the data accesses as reads and writes to a FIFO, with associated empty, full and data valid signals.

ap_bus - This interface implements pointer and pass-by-reference variables as a general purpose bus access similar to a typical DMA interface.

ap_memory - This interface is the default type for arrays arguments and can only be specified on array arguments. An **ap_memory** interface results in an RTL implementation which accesses the array elements as data values in a RAM, with associated address, chip enable and write enable control signals. The `set_directive_resource` command should be used to identify which RAM resource in the technology library is used for the array: this will in turn specify the number of ports available and which control signals are implemented.

ap_ctrl_none and **ap_ctrl_hs** - These interface types can only be specified on the function return argument. The **ap_ctrl_hs** is the default and adds function level control signals: an input start signal, output idle and done signals. If there is a function return argument, the done signal signifies when the return value is valid. The **ap_ctrl_none** type ensures these control signals are not added to the design.

Options

-all (**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_ctrl_none**|**ap_ctrl_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - The default interface type for all ports types (input, output and inout) and function-level handshakes. The default is **ap_none**.

-clock_enable - Add a clock-enable port (ap_ce) to the design. The clock enable prevents all clock operations when it is active low: disables all sequential operations.

-expose_global - Expose global variables as I/O ports. If a variable is created as a global but all read and write accesses are local to the design, the resource will be created in the design and there is no need for an IO port in the RTL. If however, the global variable is expected to be an external source or destination outside the RTL block, ports should be created using this option.

-in (**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all input (read-only) arguments. The default is **ap_none**.

-inout (**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all inout (read-write) arguments. The default is **ap_none**.

-out (**ap_none**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all output (write-only) arguments. The default is **ap_none**.

-return (**ap_ctrl_none**|**ap_ctrl_hs**) - Specify if function level handshakes are used or not. The default is **ap_ctrl_hs**.

Pragma

There is no pragma equivalent of the `config_interface` command.

Examples

Use an acknowledge interface for input ports.

```
config_interface -in ap_ack
```

Specify all outputs to use a handshake interface.

```
config_interface -out ap_hs
```

Do not implement any function level handshakes signals.

```
config_interface -return ap_ctrl_none
```

Configure all IO ports (read-write) to be implemented as valid interfaces and add a clock enable port to the design.

```
config_interface -inout ap_vld -clock_enable
```

config_rtl

Syntax

config_rtl [*OPTIONS*] <*model_name*>

Description

This configures various attributes of the output RTL, such as the type of reset used, the encoding of the state machines and allows a user specific identification to be used in the RTL.

By default, these options are applied to the top-level design and all RTL blocks within the design. Optionally, a specific RTL model may be specified.

<*model_name*> - The RTL module to configure. If none is provided, the top-level design (and all sub-blocks) is assumed.

Options

-header <*string*> - This option places the contents of file <*string*> at the top (as comments) of all output RTL and simulation files. This can be used to ensure the output RTL files contain user specified identification.

-prefix <*string*> - Specify a prefix to be added to all RTL entity/module names.

-reset (**none** | **control** | **state** | **all**) - Variables initialized in the C code are always initialized to the same value in the RTL and hence in the bit-stream. This initialization however is only performed at power-on and not repeated when a reset is applied to the design. The setting applied with the **-reset** option determines how registers/memories are reset. The default is **control**.

- **none** - no reset is added to the design.
- **control** - reset control registers, such as those used in state machines and to generate IO protocol signals.
- **state** - reset control registers and registers/memories derived from static/global variables in the C code. Any static/global variable initialized in the C code is reset to its initialized value.

- **all** - reset all registers and memories in the design. Any static/global variable initialized in the C code is reset to its initialized value.

-reset_async - This option causes all registers to use a asynchronous reset. If this option is not specified a synchronous reset is used.

-reset_level (**low**|**high**) - This option allows the polarity of the reset signal to be either active low or active high. The default is **high**.

-encoding (**bin**|**onehot**|**gray**) - Specify the encoding style used by the design's state machine. The default is **bin**.

Pragma

There is no pragma equivalent of the `config_rtl` command.

Examples

This example configures the output RTL to have all registers reset with an asynchronous active low reset.

```
config_rtl -reset all -reset_async -reset_level low
```

Add the contents of file `my_message.txt` as a comment to all RTL output files.

```
config_rtl -header my_mesage.txt
```

config_schedule

Syntax

config_schedule [*OPTIONS*]

Description

This configures the default type of scheduling performed by High-Level Synthesis.

Options

-effort (**high**|**medium**|**low**) - Specify the effort used during scheduling operations. The default is **medium** effort. A **low** effort optimization will improve the run time and may be useful for cases where when there are few choices for the design implementation. A **high** effort optimization will result in increased run time but will typically provide better quality of results.

-verbose - The verbose option will print out the critical path when scheduling fails to satisfy any directives or constraints.

Pragma

There is no pragma equivalent of the config_schedule command.

Examples

Change the default schedule effort to low to reduce run time.

```
config_schedule -effort low
```

create_clock

Syntax

```
create_clock -period <number> [OPTIONS]
```

Description

The create_clock command creates a virtual clock for the current solution. The command can only be executed in the context of an active solution. The clock period is a constraint that drives Autopilot's optimization (chaining as many operations as feasible in the given clock period).

For C and C++ designs, only a single clock is supported. For SystemC designs, multiple named clocks can be created and applied to different SC_MODULEs using the set_directive_clock command.

Options

-name <*string*> - Specify the name of the clock. If no name is given a default name is used.

-period <*number*> - Specify the clock period in ns or Mhz. If no units are specified then ns are assumed. If no period is specified a default period of 10 ns is used.

Pragma

There is no pragma equivalent of the create_clock command.

Examples

Specify a clock period of 50ns.

```
create_clock -period 50
```

This example uses the default period of 10ns to specify the clock.

```
create_clock
```

For a SystemC designs, multiple named clocks can be created (and applied using `set_directive_clock`).

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
```

To specify clock frequency in MHz:

```
create_clock -period 100MHz
```

delete_project

Syntax

delete_project *<project>*

Description

The `delete_project` command deletes the directory associated with the project.

The command checks the corresponding project directory *<project>* to ensure it is a valid High-Level Synthesis project before deleting it. If no directory *<project>* exists in the current work directory the command has no effect.

<project> - Specify the name of the project.

Pragma

There is no pragma equivalent of the `delete_project` command.

Examples

Delete project `Project_1` by removing the directory `./Project_1` and all contents.

```
delete_project Project_1
```

delete_solution

Syntax

```
delete_solution <solution>
```

Description

The `delete_solution` command removes a solution from an active project, and deletes the <*solution*> sub-directory from the project directory.

If the solution does not exist in the project directory this command has no effect.

<*solution*> - Specify the name of the solution to be deleted.

Pragma

There is no pragma equivalent of the `delete_solution` command.

Examples

Delete solution `Solution_1` from the active project by removing the sub-directory Solution_1 from the active project directory.

```
delete_solution Solution_1
```

elaborate

Syntax

```
elaborate [OPTIONS]
```

Description

The elaborate command compiles the source files and creates the High-Level Synthesis database for the active solution. The command can only be executed in the context of an active solution.

Some initial processing of functions, loops and arrays is done, based on any directives that are set.

Options

-effort (**low**|**medium**|**high**) - By default, the **medium** effort is used: this should generally provide the best balance of run time and Quality-of-Results (QoR). The **high** effort uses transformations that will lead to the best QoR but will take longer to run. A **low** effort will run faster but may result in a less optimal design and should only be used for cases where little or no optimization is possible.

Pragma

There is no pragma equivalent of the `elaborate` command.

Examples

After specifying all input source files and libraries, elaborate the design to create an internal model which can be synthesized into RTL.

```
elaborate
```

Elaborate the design using high effort.

```
elaborate -effort high
```

help

Syntax

help [*OPTIONS*] <*cmd*>

Description

When used without any <*cmd*> the `help` command lists all the High-Level Synthesis Tcl commands.

When used with an High-Level Synthesis command as an argument, the `help` command provides information on the specified command. Auto-completion using the tab key, for legal High-Level Synthesis commands, is active when typing the command argument.

Options

<*cmd*> - Name of the command to provide more help on.

Pragma

There is no pragma equivalent of the `help` command.

Examples

List the help page for command `add_file`.

```
help add_file
```

List all commands and directives used in High-Level Synthesis.

```
help
```

list_core

Syntax

`list_core` [*OPTIONS*]

Description

List all the cores in the currently loaded library. Cores are the components used to implement operations in the output RTL (such as adders, multipliers, memories).

After elaboration the operations in the RTL are represented as operators in the internal database. During scheduling operators are mapped to cores from the library to implement the RTL design. Multiple operators may be mapped on the same instance of a core, sharing the same RTL resource.

The `list_core` command allows the available operators and cores to be listed by using the relevant option:

- Operation - This shows which cores in the library can be used to implement each operation.
- Type - Lists the available cores by type, for example those which implement functional operations or those which implement memory/storage operations.

If no options are provided, the command will list all cores in the library.

The information provided by the `list_core` command can be used with the `set_directive_resource` command to implement specific operations onto specific cores.

Options

-operation (**opers**) - List the cores in the library which can implement the specified operation. The following is the list of operations:

- **add** - Addition
- **sub** - Subtraction
- **mul** - Multiplication
- **udiv** - Unsigned Division
- **urem** - Unsigned Remainder (Modulus operator)
- **srem** - Signed Remainder (Modulus operator)
- **icmp** - Integer Compare
- **shl** - Shift-Left
- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **mux** - Multiplexor
- **load** - Memory Read
- **store** - Memory Write
- **fiforead** - FIFO Read
- **fifowrite** - FIFO Write
- **fifonbread** - Non-Blocking FIFO Read
- **fifonbwrite** - Non-Blocking FIFO Write

-type (**functional_unit** | **storage** | **connector** | **interface** | **ip_block**) - This option will only list cores of the specified type.

- Function Units - Cores which implement standard RTL operations (such as add, multiply, compare)
- Storage - Cores which implement storage elements such as registers or memories.
- Connectors - Cores used to implement connectivity within the design. This included direct connections and streaming storage elements.
- Adapter - Cores which implement interfaces used to connect the top-level design when IP is generated. These interfaces are implemented in the RTL wrapper used in the IP generation flow (Xilinx® EDK).
- IP Blocks - Any IP cores added by you.

Pragma

There is no pragma equivalent of the `list_core` command.

Examples

This example lists all core in the currently loaded libraries which can implement a **`add`** operation.

```
list_core -operation add
```

Here, all available memory (storage) cores in the library are listed. The `set_directive_resource` command can be used to implement an array using one of the available memories.

```
list_core -type storage
```

list_part

Syntax

`list_part` *[OPTIONS]*

Description

This command returns the supported device families or supported parts for a given family. If no option is provided, it will return all supported families. To return parts of a family, specify one of the supported families which were listed when no option was provided to the command: this will list the parts supported within that family.

Pragma

There is no pragma equivalent of the `list_part` command.

Examples

This following example returns all supported families.

```
list_part
```

Here, all supported 'virtex6' parts are returned as a list.

```
list_part virtex6
```

open_project

Syntax

```
open_project [OPTIONS] <project>
```

Description

The open_project command opens an existing project, or creates a new one.

There can only be one project active at any given time in an High-Level Synthesis session. A project can contain multiple solutions. A project can be closed with the close_project command or by starting another project with the open_project command. The delete_project command will completely delete the project directory (removing it from the disk) and any solutions associated it.

<project> - Specify the name of the project.

Options

-reset - Resets the project by removing any project data which already exists. This option should be used when executing High-Level Synthesis with Tcl scripts, otherwise each new add_file or add_library command will add additional files to the existing data.

Any previous project information on design source files, header file search paths and the top level function is removed. The associated solution directories and files are kept (but may now have invalid results: the delete_project command will accomplish the same as the **-reset** option and remove all solution data).

Pragma

There is no pragma equivalent of the open_project command.

Examples

Open a new or existing project named Project_1.

```
open_project Project_1
```

Open a project and remove any existing data (preferred method when using Tcl scripts, to prevent adding source or library files to the existing project data).

```
open_project -reset Project_2
```

open_solution

Syntax

```
open_solution [OPTIONS] <solution>
```

Description

The `open_solution` command opens an existing solution or creates a new one in the currently active project. Trying to open or create a solution when there is no active project results in an error. There can only be one solution active at any given time in an High-Level Synthesis session.

Each solution is managed in a sub-directory of the current project directory. If the solution does not exist yet in the current work directory, then a new solution is created. A solution can be closed by the `close_solution` command or by opening another solution with the `open_solution` command. The `delete_solution` command will remove them from the project and delete the corresponding subdirectory.

`<solution>` - Specify the name of the solution.

Options

`-reset` - Resets the solution data if the solution already exists. Any previous solution information on libraries, constraints and directives is removed. Synthesis, verification and implementation results are also removed.

Pragma

There is no pragma equivalent of the `open_solution` command.

Examples

Open a new or existing solution in the active project named `Solution_1`.

```
open_solution Solution_1
```

Open a solution in the active project and remove any existing data (preferred method when using Tcl scripts, to prevent adding to the existing solution data).

```
open_solution -reset Solution_2
```

set_clock_uncertainty

Syntax

set_clock_uncertainty <*uncertainty*> <*clock_list*>

Description

The `set_clock_uncertainty` command sets a margin on the clock period defined by `create_clock`. The margin is subtracted from the clock period to create an effective clock period. If the clock uncertainty is not defined, it will default to 12.5% of the clock period.

High-Level Synthesis will optimize the design based on the effective clock period, providing a margin for downstream tools to account for logic synthesis and routing. The command can only be executed in the context of an active solution. High-Level Synthesis will still use the specified clock period in all output files for verification and implementation.

For SystemC designs where multiple named clocks are specified by the `create_clock` command, a different clock uncertainty can be specified on each named clock by specifying the named clock.

<*uncertainty*> - A value, specified in ns, which represents how much of the clock period is used as a margin.

<*clock_list*> - A value, specified in ns, which represents how much of the clock period is used as a margin.

Pragma

There is no pragma equivalent of the `set_clock_uncertainty` command.

Examples

Specify an uncertainty/margin of 0.5ns on the clock: this effectively reduces the clock period High-Level Synthesis can use by 0.5ns.

```
set_clock_uncertainty 0.5
```

In this SystemC example, two clock domains and created and a different clock uncertainty is specified on each domain. (Multiple clocks are supported in SystemC designs: the `set_directive_clock` command is used to apply the clock to the appropriate function).

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
set_clock_uncertainty 0.5 fast_clock
```

```
set_clock_uncertainty 1.5 slow_clock
```

set_directive_allocation

Syntax

set_directive_allocation [*OPTIONS*] <*location*> <*instances*>

Description

Specify instance restrictions for resource allocation. This defines, and can limit, the number of RTL instances used to implement specific functions or operations.

For example, if the C source has four instances of a function `foo_sub`, the `set_directive_allocation` command can be used to ensure there is only one instance of `foo_sub` in the final RTL (all four instances will be implemented using the same RTL block).

<*location*> - The location string in the format of function[/label].

<*instances*> - A function or operator.

The function can be any function in the original C code and which has not been inlined by the `set_directive_inline` command or inlined automatically by High-Level Synthesis.

The list of operators is as follows (provided there is an instance of such an operation in the C source code):

- **add** - Addition
- **sub** - Subtraction
- **mul** - Multiplication
- **icmp** - Integer Compare
- **sdiv** - Signed Division
- **udiv** - Unsigned Division
- **srem** - Signed Remainder
- **urem** - Unsigned Remainder
- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **shl** - Shift-Left

Options

-limit <*integer*> - A maximum limit on the number of instances (of the type defined by the **-type** option) to be used in the RTL design.

-type (**function** | **operation**) - The instance type can be **function** or **operation**. The default is **function**.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP allocation \
        instances=<Instance Name List> \
        limit=<Integer Value> \
        <operation, function>
```

Examples

Given a design `foo_top` with multiple instances of function `foo`, this command (the pragma is also shown) limits the number of instances of `foo` in the RTL to 2.

```
set_directive_allocation -limit 2 -type function foo_top foo
#pragma AP allocation instances=foo limit=2 function
```

The following command (the pragma is also shown) limits the number of multipliers used in the implementation of `My_func` to 1.

Note: This limit does not apply to any multipliers which may reside in sub-functions of `My_func`. To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `My_func`.

```
set_directive_allocation -limit 1 -type operation My_func mul
#pragma AP allocation instances=mul limit=1 operation
```

set_directive_array_map

Syntax

set_directive_array_map [*OPTIONS*] <*location*> <*array*>

Description

This command maps a smaller array into a larger array. The typical usage is to use multiple `set_directive_array_map` commands (with the same -instance target) to map multiple smaller arrays into a single larger array which can then be targeted to a single larger memory (RAM or FIFO) resource.

The **-mode** option is used to determine if the new target is a concatenation of elements (horizontal mapping) or bit-widths (vertical mapping). The arrays are concatenated in the order the `set_directive_array_map` commands are issued starting at target element zero in horizontal mapping or bit zero in vertical mapping.

<*location*> - The name of the location, in the format function[/label], which contains the array variable.

<*variable*> - Specify the name of the array variable to be mapped into the new target array instance.

Options

-instance <*string*>- Specify the new array instance name, where the current array variable is to be mapped.

-mode (**horizontal** | **vertical**) - Horizontal mapping concatenates the arrays to form a target with more elements. Vertical mapping concatenates the array to form a target with longer words. The default is **horizontal**.

-offset <*integer*> - This is only relevant for horizontal mapping and specifies an integer value that indicates the absolute offset in the target instance for current mapping operation: element 0 of the array variable will map to element <*int*> of the new target (for example, other elements will map to <*int+1*>, <*int+2*>). If the value is not specified, High-Level Synthesis will calculate the required offset automatically, to avoid any overlap (for example, concatenate the arrays starting at the next unused element in the target).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_map \
        variable=<variable> \
        instance=<instance> \
        <horizontal, vertical> \
        offset=<int>
```

Examples

The following commands (the equivalent pragmas are also shown) map arrays A[10] and B[15] in function `foo` into a single new array AB[25]. Element AB[0] will be the same as A[0], element AB[10] will be the same as B[0] (since no -offset option is used) and the bit-width of array AB[25] will be the maximum bit-width of A[10] or B[15].

```
set_directive_array_map -instance AB -mode horizontal foo A
set_directive_array_map -instance AB -mode horizontal foo B
#pragma AP array_map variable=A instance=AB horizontal
#pragma AP array_map variable=B instance=AB horizontal
```

This example concatenates arrays C and D into a new array CD with same number of bits as C and D combined. The number of elements in CD will be maximum of C or D.

```
set_directive_array_map -instance CD -mode vertical foo C
set_directive_array_map -instance CD -mode vertical foo D
#pragma AP array_map variable=C instance=CD vertical
#pragma AP array_map variable=D instance=CD vertical
```

set_directive_array_partition

Syntax

set_directive_array_partition [*OPTIONS*] <*location*> <*array*>

Description

Partitions an array into smaller arrays or individual elements. This will result in RTL with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage, potentially improving the throughput of the design, but will require more memory instances or registers.

<*location*> - The name of the location, in the format function[/label], which contains the array variable.

<*array*> - Specify the name of the array variable to be partitioned.

Options

-dim <*integer*> - This is only relevant for multi-dimensional arrays and specifies which dimension of the array is to be partitioned. If a value of 0 is used, all dimensions will be partitioned with the specified options. Any other value will partition only that dimension, for example, if a value 1 is used, only the first dimension will be partitioned.

-factor <*integer*> - Integer number to specify the number of smaller arrays which are to be created. This option is only relevant for type **block** or **cyclic** partitioning.

-type (**block**|**cyclic**|**complete**) - Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the -factor option. The default is **complete**.

Cyclic partitioning creates smaller arrays by interleaving elements from the original array. For example, if **-factor** 3 is used, element 0 is assigned to the first new array, element 1 to the second new array, element 3 is assigned to the third new array and then element 4 is assigned to the first new array again.

Complete partitioning decomposes the array into individual elements. For a one-dimensional array this corresponds to resolving a memory into individual registers. For multi-dimensional arrays, specify the partitioning of each dimension or use **-dim** 0 to partition all dimensions.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_partition \
        variable=<variable> \
        <block, cyclic, complete> \
        factor=<int> \
        dim=<int>
```

Examples

The following command (the equivalent pragma is also shown) partitions array AB[13] in function `foo` into four arrays. Because 4 is not an integer multiple of 13, three of the arrays will have 3 elements and one will have 4 (containing elements AB[9:12]).

```
set_directive_array_partition -type block -factor 4 foo AB
#pragma AP array_partition variable=AB block factor=4
```

This example partitions array AB[6][4] in function `foo` into two arrays, each of dimension [6][2].

```
set_directive_array_partition -type block -factor 2 -dim 2 foo AB
#pragma AP array_partition variable=AB block factor=2 dim=2
```

All dimensions of AB[4][10][6] in function `foo` are partitioned into individual elements by this command.

```
set_directive_array_partition -type complete -dim 0 foo AB
#pragma AP array_partition variable=AB complete dim=0
```

set_directive_array_reshape

Syntax

set_directive_array_reshape [*OPTIONS*] <*location*> <*array*>

Description

This command combines array partitioning with vertical array mapping, to create a single new array with fewer elements but wider words.

It will first split the array into multiple arrays (in an identical manner as `set_directive_array_partition`) and then automatically recombine the arrays vertically (as per `set_directive_array_map` **`-type`** vertical) to create a new array with wider words.

<*location*> -The name of the location, in the format function[/label], which contains the array variable.

<*array*> - Specify the name of the array variable to be reshaped.

Options

`-dim` <*integer*> - This is only relevant for multi-dimensional arrays and specifies which dimension of the array is to be reshaped. If a value of 0 is used, all dimensions will be partitioned with the specified options. Any other value will partition only that dimension, for example, if a value 1 is used, only the first dimension will be partitioned.

`-factor` <*integer*> - Integer number to specify the number of temporary smaller arrays which are to be created; Only relevant for type **`block`** or **`cyclic`** reshaping.

`-type` (**`block`**|**`cyclic`**|**`complete`**) - Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the -factor option and then combines the N blocks into a single array with word-width*N. The default is **`complete`**.

Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if **`-factor`** 3 is used, element 0 is assigned to the first new array, element 1 to the second new array, element 3 is assigned to the third new array and then element 4 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.

Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with N*M bits).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_reshape \
        variable=<variable> \
        <block, cyclic, complete> \
        factor=<int> \
        dim=<int>
```

Examples

The following command (the equivalent pragma is also shown) reshapes 8-bit array AB[17] in function `foo`, into a new 32-bit array with 5 elements. Since 4 is not an integer multiple of 13, AB[17] will be in the lower 8-bits of the 5th element (the remainder of the 5th element is unused).

```
set_directive_array_reshape -type block -factor 4 foo AB
#pragma AP array_reshape variable=AB block factor=4
```

This example partitions array AB[6][4] in function `foo`, into a new array of dimension [6][2], where dimension 2 is twice the width.

```
set_directive_array_reshape -type block -factor 2 -dim 2 foo AB
#pragma AP array_reshape variable=AB block factor=2 dim=2
```

This command reshapes 8-bit array AB[4][2][2] in function `foo`, into a new single element array (a register), 4*2*2*8(=128)-bits wide.

```
set_directive_array_reshape -type complete -dim 0 foo AB
#pragma AP array_reshape variable=AB complete dim=0
```

set_directive_array_stream

Syntax

set_directive_array_stream [*OPTIONS*] <*location*> <*variable*>

Description

By default, array variables are implemented as RAM (random access) memories:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- In sub-functions involved in dataflow optimizations, the array arguments are implemented using a RAM ping-pong buffer channel.
- Arrays involved in loop-based dataflow optimizations are implemented as a RAM ping-pong buffer channel.

However, if the data stored in the array is consumed/produced in a sequential manner, a more efficient communication mechanism is to use streaming data, where FIFOs are used instead of RAMs.

Note: When an argument of the top-level function is specified as interface type `ap_fifo`, the array is automatically identified as streaming.

`<location>` - The name of the location, in the format function[/label], which contains the array variable.

`<variable>` - Name of the array variable to be implemented as a FIFO.

Options

`-depth` `<integer>` - Only relevant for array streaming in dataflow channels, this is used to override the default FIFO depth specified (globally) by the `config_dataflow` command.

`-off` - This option is only relevant for array streaming in dataflow channels. If used, the `config_dataflow -default_channel fifo` command globally implies a `set_directive_array_stream` on all arrays in the design. This option allows streaming to be turned off on a specific array (and default back to using a RAM ping-pong buffer based channel).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_stream
        variable=<variable> \
        off \
        depth=<int>
```

Examples

The following command (the equivalent pragma is also shown) specifies array A[10] in function `foo` to be streaming, and implemented as a FIFO.

```
set_directive_array_stream foo A
#pragma AP array_reshape variable=A
```

In this example, array B in named loop `loop_1` of function `foo`, is set to streaming with a FIFO depth of 12. In this case, the pragma should be placed inside `loop_1`.

```
set_directive_array_stream -depth 12 foo/loop_1 B
#pragma AP array_reshape variable=B depth=12
```

Here, array C has streaming disabled (it's assumed enabled by `config_dataflow` in this example)

```
set_directive_array_stream -off foo C
#pragma AP array_reshape variable=C off
```

set_directive_clock

Syntax

`set_directive_clock` *<location> <domain>*

Description

Applies the named clock to the specified function.

In C and C++ designs, only a single clock is supported and the clock period specified by `create_clock` is automatically applied to all functions in the design.

SystemC designs support multiple clocks. Multiple named clocks may be specified using the `create_clock` command and applied to individual SC_MODULEs using the `set_directive_clock` command. Each SC_MODULE is synthesized using a single clock.

<location> - The name of the function where the named clock is to be applied.

<domain> - The name of the clock, as specified by the **`-name`** option of the `create_clock` command.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP clock domain=<string>
```

Examples

Given a SystemC design, where top-level `foo_top` has clocks ports `fast_clock` and `slow_clock` but only uses `fast_clock` within its function and sub-block `foo` which only uses `slow_clock`, the following commands create both clocks, apply `fast_clock` to `foo_top` and `slow_clock` to sub-block foo. The equivalent pragmas are also shown and should be placed in the scope of the appropriate function:

Note: There is no pragma equivalent of `create_clock`.

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk

set_directive_clock foo_top fast_clock
set_directive_clock foo slow_clock
#pragma AP clock domain=fast_clock
#pragma AP clock domain=slow_clock
```

set_directive_dataflow

Syntax

set_directive_dataflow [*OPTIONS*] <*location*>

Description

The `set_directive_dataflow` command specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation.

In a C description, all operations are performed in a sequential manner. High-Level Synthesis automatically seeks to minimize latency and improve concurrency (in the absence of any directives which limit resources, such as `set_directive_allocation`) however data dependencies can limit this. For example, functions or loops which access arrays must finish all read/write accesses to the arrays before they complete, preventing the next function or loop, which consumes the data, from starting operation.

It may however be possible for the operations in a function or loop to start execution before the previous function or loop completes all its operations.

When dataflow optimization is specified, High-Level Synthesis will analyze the dataflow between sequential functions or loops, and seek to create channels (based on ping-pong RAMs or FIFOs) which allow consumer functions or loops to start operation before the producer functions or loops have completed, enabling functions or loops to operate in parallel, decreasing the latency and improving the throughput of the RTL design.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, High-Level Synthesis will seek to minimize the initiation interval and start operation as soon as data is available.

<*location*> - The name of the location, in the format function[/label], where dataflow optimization is to be performed.

Options

-interval <*integer*> - An integer value specifying the desired initiation interval (II): the number of cycles between the first function or loop executing and the start of execution of the next function or loop.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP dataflow interval=<int>
```

Examples

This example specifies dataflow optimization within function `foo`. The equivalent pragma is also shown.

```
set_directive_dataflow foo
#pragma AP dataflow interval=3
```

In this example, dataflow is specified in function `My_Func`, with a target initiation interval of 3.

```
set_directive_dataflow -interval 3 My_func
#pragma AP dataflow interval=3
```

set_directive_data_pack

Syntax

```
set_directive_data_pack [OPTIONS] <location> <variable>
```

Description

This directive packs the data fields of a struct into a single scalar with a wider word width. Any arrays declared inside the struct will be completely partitioned and reshaped into a wide scalar and be packed with other scalar fields.

The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct fields. The first field takes the least significant sector of the word and so forth until all fields are mapped.

<*location*> - The name of the location, in the format function[/label], which contains the variable which will be packed.

<*variable*> - Specify the name of the variable to be packed.

Options

-instance <*string*> - Specify the name of resultant variable after packing. If none is provided, the name of the input `variable` will be used.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP data_pack variable=<variable> instance=<string>
```

Examples

The following command (the equivalent pragma is also shown) packs struct array AB[17] with three 8-bit field fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 17 element array of 24-bits.

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

This example (the equivalent pragma is also shown) packs struct pointer AB with three 8-bit fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 24-bit pointer.

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

set_directive_dependence

Syntax

set_directive_dependence [*OPTIONS*] <*location*>

Description

High-Level Synthesis automatically detects dependencies within loops (a loop-independent dependency) or between different iterations of a loop (a loop-carry dependency). These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

Loop-independent dependence: the same element gets accessed in the same loop iteration.

```
for (i=0;i<N;i++) {
       A[i]=x;
       y=A[i];
}
```

Loop-carry dependence: the same element gets accessed in a different loop iteration.

```
for (i=0;i<N;i++) {
       A[i]=A[i-1]*2;
}
```

Under certain circumstances such as variable dependent array indexing or when an external requirement needs enforced (for example, two inputs are never the same index) the dependence analysis may be too conservative. The `set_directive_dependence` command helps allows you to explicitly specify the dependence and resolve a false dependence.

<*location*> - The name of the location, in the format function[/label], where the dependence is to be specified.

Options

-class (**array**|**pointer**) - Specify a class of variables where the dependence need clarification. This is mutually exclusive with the option **-variable**.

-dependent (**true**|**false**) - Specify if a dependence needs to be enforced (true) or removed (false). The default is **false**.

-direction (**RAW**|**WAR**|**WAW**) - This is only relevant for loop-carry dependencies. Specify the direction for a dependence:

- **RAW** (Read-After-Write - true dependence) - the write instruction uses a value used by the read instruction.
- **WAR** (Write-After-Read - anti dependence) - the read instruction gets a value that is overwritten by the write instruction.
- **WAW** (Write-After-Write - output dependence) - two write instructions write to the same location, in a certain order.

-distance <*integer*> - This is only relevant for loop-carry dependencies where option **-dependent** is set to `true`. A positive integer to specify the inter-iteration distance for array access.

-type (**intra**|**inter**) - Specify if the dependence is within the same loop iteration (intra) or between different loop iterations (inter). The default is **inter**.

-variable <*variable*> - Specify the specific variable to consider for the dependence directive. This is mutually exclusive with the option **-class**.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP dependence \
        variable=<variable> \
        <array, pointer> \
        <inter, intra> \
        <RAW, WAR, WAW> \
        distance=<int> \
        <false, true>
```

Examples

This example removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`. The equivalent pragma is also shown.

```
set_directive_dependence -variable Var1 -type intra \
-dependent false foo/loop_1
#pragma AP dependence variable=Var1 intra false
```

Here, the dependence on all arrays in `loop_2` of function foo, is set to inform High-Level Synthesis all reads must happen after writes in the same loop iteration.

```
set_directive_dependence -class array -type inter \
-dependent true -direction RAW foo/loop_2
#pragma AP dependence array inter RAW true
```

set_directive_expression_balance

Syntax

`set_directive_expression_balance` [*OPTIONS*] <*location*>

Description

Sometimes, a C-based specification is written with a sequence of operations. This can result in a lengthy chain of operations in RTL, and with a small clock period, this could increase the design latency.

By default, High-Level Synthesis will rearrange the operations, through associative and commutative properties, to create a balanced tree which can shorten the chain, potentially reducing latency at the cost of extra hardware.

The `set_directive_expression_balance` command allows this expression balancing to be turned off or on within with a specified scope.

<*location*> - The name of the location, in the format function[/label], where the balancing should be enabled or disabled.

Options

`-off` - Turn off expression balancing at this location.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP expression_balance <off>
```

Examples

This example disables expression balancing within function `My_Func`. The equivalent pragma is also shown.

```
set_directive_expression_balance -off My_Func
#pragma AP expression_balance off
```

Conversely, this example explicitly enables expression balancing in function `My_Func2`.

```
set_directive_expression_balance My_Func2
#pragma AP expression_balance
```

set_directive_function_instantiate

Syntax

`set_directive_function_instantiate` *<location> <variable>*

Description

By default,

- Functions remain as separate hierarchy blocks in the RTL.
- All instances of a function, at the same level of hierarchy, will use the same RTL implementation (block).

The **`set_directive_function_instantiate`** command is used to create a unique RTL implementation for each instance of a function, allowing each instance to be optimized.

By default, the following code would result in a single RTL implementation of function `foo_sub` for all three instances.

```
char foo_sub(char inval, char incr)
{
       return inval + incr;
}
```

```
void foo(char inval1, char inval2, char inval3,
       char *outval1, char *outval2, char * outval3)
{
       *outval1 = foo_sub(inval1, 1);
       *outval2 = foo_sub(inval2, 2);
       *outval3 = foo_sub(inval3, 3);
}
```

Using the directive, in the manner shown in the example section below, would result in three versions of function `foo_sub`, each independently optimized for variable `incr`.

<*location*> - The name of the location, in the format function[/label], where the instances of a function are to be made unique.

variable <*string*> - Specify which function argument <*string*> is to be specified as constant.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP function_instantiate variable=<variable>
```

Examples

For the example code shown above, the following Tcl (or pragma placed in function `foo_sub`) allows each instance of function `foo_sub` to be independently optimized with respect to input **`incr.`**

```
set_directive_function_instantiate incr foo_sub
#pragma AP function_instantiate variable=incr
```

set_directive_inline

Syntax

set_directive_inline [*OPTIONS*] <*location*>

Description

This command removes a function as a separate entity in the hierarchy. After inlining the function will be dissolved and no longer appear as a separate level of hierarchy.

In some cases inlining a function will allow operations within the function to be shared and optimized more effectively with surrounding operations. An inlined function however, cannot be shared: in some cases this may increase area.

By default, inlining is only performed on the next level of function hierarchy.

<location> - The name of the location, in the format function[/label], where inlining is to be performed.

Options

-off - This disables function inlining and is used to prevent particular functions from being inlined. For example, if the **-recursive** option is used in a caller function, this option can prevent a particular callee function from being inlined when all others are.

-recursive - By default only one level of function inlining is performed: the functions within the specified function are not inlined. The **-recursive** option inlines all functions recursively down the hierarchy.

-region - All functions in the specified region are to be inlined.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP inline <region | recursive | off>
```

Examples

This example inlines all functions in `foo_top` (but not any lower level functions).

```
set_directive_inline -region foo_top
#pragma AP inline region
```

Here, only function `foo_sub1` is inlined.

```
set_directive_inline foo_sub1
#pragma AP inline
```

These commands inline all functions in `foo_top`, recursively down the hierarchy, except function `foo_sub2`. The first pragma is placed in function `foo_top`. The second pragma is placed in function `foo_sub2`.

```
set_directive_inline -region -recursive foo_top
set_directive_inline -off foo_sub2
```

```
#pragma AP inline region recursive
#pragma AP inline off
```

set_directive_interface

Syntax

set_directive_interface [*OPTIONS*] <*location*> <*port*>

Description

The set_directive_interface command specifies how RTL ports are created from the function description during interface synthesis.

The ports in the RTL implementation are derived from:

- Any function-level protocol which is specified.
- Function arguments.
- Global variables - accessed by the top-level function and defined outside its scope.

Function-level handshakes are used to control when the function starts operation and to indicate when function operation ends, is idle and when (in the case of a pipelined function) it is ready for new inputs. The implementation of a function-level protocol is controlled by modes **ap_ctrl_none** or **ap_ctrl_hs** and only requires the top-level function name (the function return should be specified for the pragma).

Each function argument can be specified to have its' own IO protocol (such as valid handshake, acknowledge handshake).

If a global variable is accessed but all read and write operations are local to the design, the resource will be created in the design and there is no need for an IO port in the RTL. If however, the global variable is expected to be an external source or destination it should have its' interface specified in a similar manner as standard function arguments (refer to the examples section).

When set_directive_interface is used on sub-functions only the **-register** option can be used: the **-mode** option is not supported on sub-functions.

<*location*> - The name of the location, in the format function[/label], where the function interface or registered output is to be specified.

<*port*> - The parameter (function argument or global variable) for which the interface has to be synthesized. This is not required when modes **ap_ctrl_none** or **ap_ctrl_hs** are used.

Options

-mode (**ap_ctrl_none**|**ap_ctrl_hs**|**ap_none**|**ap_stable**|**ap_vld**|**ap_ovld**|**ap_ack**|**ap_hs**|**ap_fifo**|**ap_memory**|**ap_bus**) - Select the appropriate protocol.

Function protocol is implemented by the following **-mode** values:

- **ap_ctrl_none** - No function-level handshake protocol.
- **ap_ctrl_hs** - This is the default behavior and implements a function-level handshake protocol. Input port `ap_start` must go high for the function to begin operation. (All function-level signals are active high). Output port `ap_done` indicates the function is finished (and if there is a function return value, indicates when the return value is valid) and output port `ap_idle` indicates when the function is idle. In pipelined functions, an additional output port `ap_ready` is implemented and indicates when the function is ready for new input data.

For function arguments and global variables, the following default protocol is used for each argument type:

- Read-only (Inputs) - **ap_none**
- Write-only (Outputs) - **ap_vld**
- Read-Write (Inouts) - **ap_ovld**
- Arrays - **ap_memory**

The RTL ports to implement function arguments and global variables are specified by the following **-mode** values:

- **ap_none** - No protocol in place. This corresponds to a simple wire.
- **ap_stable** - Only applicable to input ports, this informs High-Level Synthesis that the value on this port is stable after reset and is guaranteed not to change until the next reset. The protocol is implemented as mode **ap_none** but this allows internal optimizations to take place on the signal fanout.

  Note: This is not considered a constant value, simply an unchanging value.
- **ap_vld** - An additional valid port is created (`<port_name>_vld`) to operate in conjunction with this data port. For input ports a read will stall the function until its associated input valid port is asserted. An output port will have its output valid signal asserted when it writes data.
- **ap_ack** - An additional acknowledge port is created (`<port_name>_ack`) to operate in conjunction with this data port. For input ports, a read will assert the output acknowledge when it reads a value. An output write will stall the function until its associated input acknowledge port is asserted.
- **ap_hs** - Additional valid (`<port_name>_vld`) and acknowledge (`<port_name>_ack`) ports are created to operate in conjunction with this data port. For input ports a

read will stall the function until its input valid is asserted and will assert its output acknowledge signal when data is read. An output write will assert an output valid when it writes data and stall the function until its associated input acknowledge port is asserted.

- **ap_ovld** - For input signals, this acts as mode **ap_none** and no protocol is added. For output signals, this acts as mode **ap_vld**. For inout signals, the input gets implemented as mode **ap_none** and the output as mode **ap_vld**.
- **ap_memory** - This mode implements array arguments as accesses to an external RAM. Data, address and RAM control ports (such as CE, WE) are created to read from and write the external RAM. The specific signals and number of data ports are determined by the RAM which is being accessed. The array argument should be targeted to a specific RAM in the technology library using the `set_directive_resource command` (or High-Level Synthesis will automatically determine the RAM to use).
- **ap_fifo** - Implements array, pointer and pass-by-reference ports as a FIFO access. The data input port will assert its associated output read port (`<port_name>_read`) when it is ready to read new values from the external FIFO and will stall the function until its input available port (`<port_name>_empty_n`) is asserted to indicate a value is available to read. An output data port will asset an output write port (`<port_name>_write`) to indicate it has written a value to the port and will stall the function until its associated input available port (`<port_name>_full_n`) is asserted to indicate there is space available in the external FIFO for new outputs. This interface mode should use the **-depth** option.
- **ap_bus** - implements pointer and pass-by-reference ports as a bus interface. Both input and output ports are synthesized with a number of control signals to support burst access to and from a standard FIFO bus interface. Refer to the *High-Level Synthesis User Guide (UG867)* for a detailed description of this interface. This interface mode should use the –depth option.

-depth - The depth option is required for pointer interfaces using **ap_fifo** or **ap_bus** modes. This option should be used to specify the maximum number of samples which will be processed by the testbench. This is required to inform High-Level Synthesis about the maximum size of FIFO needed in the verification adapter created for RTL co-simulation.

-register - For the top-level function, this option is relevant for scalar interfaces **ap_none**, **ap_ack**, **ap_vld**, **ap_ovld**, **ap_hs** and causes the signal (and any relevant protocol signals) to be registered and persist until at least the last cycle of the function execution. This option requires the `ap_ctrl_hs` function protocol to be enabled. If this option is used with `ap_ctrl_hs` is results in the function return value being registered.

This option can be used in sub-functions to register the outputs and any control signals until the end of the function execution.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP interface <mode> register port=<string>
```

Examples

This example turns off function-level handshakes for function `foo`.

```
set_directive_interface -mode ap_ctrl_none foo
#pragma AP interface ap_ctrl_none port=return
```

Here, argument InData in function `foo` is specified to have a **`ap_vld`** interface and the input should be registered.

```
set_directive_interface -mode ap_vld -register foo InData
#pragma AP interface ap_vld register port=InData
```

This example exposes global variable `lookup_table` used in function `foo` as a port on the RTL design, with an **`ap_memory`** interface.

```
set_directive_interface -mode ap_memory foo look_table
```

set_directive_latency

Syntax

`set_directive_latency` [*OPTIONS*] <*location*>

Description

Allows a maximum and/or minimum latency value to be specified on a function, loop or region. High-Level Synthesis will always aim for minimum latency. The behavior of High-Level Synthesis when minimum and maximum latency values are specified is explained below.

- Latency is less than the minimum - If High-Level Synthesis can achieve less than the minimum specified latency, it will extend the latency to the specified value and potentially increasing sharing.
- Latency is greater than the minimum - The constraint is satisfied and no further optimizations are performed.

- Latency is less than the maximum - The constraint is satisfied and no further optimizations are performed.
- Latency is greater than the maximum- -If High-Level Synthesis cannot schedule within the maximum limit it will automatically increase effort to achieve the specified constraint. If it still fails to meet the maximum latency a warning is issued and High-Level Synthesis will proceed to produce a design with the smallest achievable latency.

<*location*> - The name of the location (function, loop or region), in the format function[/label], to be constrained.

Options

-max <*integer*> - An integer value specifying the maximum latency.

-min <*integer*> - An integer value specifying the minimum latency.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP latency \
        min=<int> \
        max=<int>
```

Examples

In this example, function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8.

```
set_directive_latency -min=8 -max=8 foo
#pragma AP latency min=4 max=4
```

In function `foo`, loop `loop_row` is specified to have a maximum latency of 12. The pragma should be placed in the loop body.

```
set_directive_latency -max=12 foo/loop_row
#pragma AP latency max=12
```

set_directive_loop_flatten

Syntax

```
set_directive_loop_flatten [OPTIONS] <location>
```

Description

This command is used to flatten nested loops into a single loop hierarchy. In the RTL implementation it will cost a clock cycle to move between loops in the loop hierarchy and flattening nested loops allows them to be optimized as a single loop, saving clock cycles and potentially allowing greater optimization of the loop-body logic.

This directive should be applied to the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner.

- Perfect loop nest - only the inner-most loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.
- Semi-perfect loop nest - only the inner-most loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

<location> - The name of the location (inner-most loop), in the format function[/label].

Options

-off - This option prevents flattening from taking place. This can be used to prevent some loops from being flattened while all others in the specified location are flattened.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_flatten off
```

Examples

This example flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. The pragma should be placed in the body of `loop_1`.

```
set_directive_loop_flatten foo/loop_1
#pragma AP loop_flatten
```

Here, loop flattening is prevented in `loop_2` of function `foo`. The pragma should be placed in the body of `loop_2` .

```
set_directive_loop_flatten -off foo/loop_2
#pragma AP loop_flatten off
```

set_directive_loop_merge

Syntax

set_directive_loop_merge *<location>*

Description

Merge all the loops into a single loop. Merging loops reduces the number of clock cycles required in the RTL to transition between the loop-body implementations) and allows the loops be implemented in parallel (if possible).

The rules for loop merging are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO reads: merging would change the order of the reads and reads from a FIFO or FIFO interface must always be in sequence.

<location> - The name of the location, in the format function[/label], where the loops reside.

Options

-force - This option forces loops to be merged even when High-Level Synthesis issues a warning. In this case user takes responsibility that the merged loop will function correctly.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_merge force
```

Examples

This example merges all consecutive loops in function `foo` into a single loop.

```
set_directive_loop_merge foo
#pragma AP loop_merge
```

In this example, all loops inside `loop_2` (but not `loop_2` itself) of function `foo` are merged by using the **-force** option. The pragma should be placed in the body of `loop_2`.

```
set_directive_loop_merge -force foo/loop_2
#pragma AP loop_merge force
```

set_directive_loop_tripcount

Syntax

set_directive_loop_tripcount [*OPTIONS*] <*location*>

Description

The total number of iterations performed by a loop is referred to as the loop tripcount. High-Level Synthesis reports the total latency of each loop: the number of cycles to execute all iterations of the loop. This loop latency is therefore a function of the tripcount (number of loop iterations).

The tripcount could be a constant value, may depend on the value of variables used in the loop expression (for example, x<y) or control statements used inside the loop. In some cases, such as when the variables used to determine the tripcount are input arguments or variables calculated by dynamic operation, High-Level Synthesis may not be able to determine the tripcount and hence the loop latency might be unknown.

To help with the design analysis which drives optimization, the `set_directive_loop_tripcount` command allows you to specify minimum, average and maximum tripcounts for a loop and see how the loop latency contributes to the total design latency in the reports.

<*location*> - The name of the location of the loop, in the format function[/label], where the tripcount needs to be specified.

Options

-avg <*integer*> - Specify the average latency.

-max <*integer*> - Specify the maximum latency.

-min <*integer*> - Specify the minimum latency.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_tripcount \
        min=<int> \
        max=<int> \
        avg=<int>
```

Examples

In this example, `loop_1` in function `foo` is specified to have a minimum tripcount of 12, an average of 14 and maximum of 16.

```
set_directive_loop_tripcount -min 12 -max 14 -avg 16 foo/loop_1
#pragma AP loop_tripcount min=12 max=14 avg=16
```

set_directive_loop_unroll

Syntax

set_directive_unroll [*OPTIONS*] <*location*>

Description

Transforms loop by creating multiples copies of the loop body.

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by any logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic, which represents the loop-body, which is executed for the same number of iterations.

The `set_directive_loop_unroll` command allows the loop to be fully unrolled, creating as many copies of the loop-body in the RTL as there are loop iterations, or partially unrolled by a factor N, creating N copies of the loop body and adjusting the loop iteration accordingly.

If the factor N used for partial unrolling is not an integer multiple of the original loop iteration count, the original exit condition needs to be checked after each unrolled fragment of the loop body.

To unroll a loop completely, the loop bounds need to be known at compile time. This is not required for partial unrolling.

<*location*> - The location of the loop, in the format function[/label], to be unrolled.

Options

-factor <*integer*> - Non-zero integer indicating that partial unrolling is requested. The loop body will be repeated this number of times, and the iteration information will be adjusted accordingly.

-region - This option should be specified when seeking to unroll all loops within a loop without unrolling the enclosing loop itself.

Take the example where loop `loop_1` contains multiple loops at the same level of loop hierarchy, loops `loop_2` and `loop_3`. A named loop, such as `loop_1` is also a region/location in the code: a section of code enclosed by braces **{}**. If the unroll directive is specified on location <*function*>`/loop_1`, it will unroll `loop_1`.

The **-region** option specifies that the directive be applied only to the loops enclosed the named region: this results in `loop_1` being left rolled, but all loops inside it (`loop_2` and `loop_3`) being unrolled.

-skip_exit_check - This option is only effective if a factor is specified (partial unrolling).

- Fixed bounds - No exit condition check is performed if the iteration count is a multiple of the factor. For cases where the iteration count is not an integer multiple of the factor, unrolling will be prevented and a warning issued (the exit check must be performed in order to proceed).

- Variable bounds - The exit condition check is removed. The designer is responsible for ensuring the variable bounds is an integer multiple of the factor and that no exit check is in fact required.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP unroll \
        skip_exit_check \
        factor=<int> \
        region
```

Examples

This example unrolls loop `L1` in function `foo`. The pragma should be placed in the body of loop `L1`.

```
set_directive_loop_unroll foo/L1
#pragma AP unroll
```

In this example, an unroll factor of 4 is specified on loop `L2` of function `foo` and the exit check is removed. The pragma should be placed in the body of loop `L2`.

```
set_directive_loop_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

Here, all loops inside loop `L3` in function `foo` are unrolled, but not loop `L3` itself. The **`-region`** option specifies the location be considered an enclosing region and not a loop label.

```
set_directive_loop_unroll -region foo/L3
#pragma AP unroll region
```

set_directive_occurrence

Syntax

`set_directive_occurrence` [*OPTIONS*] <*location*>

Description

Used when pipelining functions or loops, to specify that the code in a location in executed at a lesser rate than the code in the enclosing function or loop. This allows the code which is executed at the lesser rate to be pipelined a slower rate and potentially shared within the top-level pipeline.

For example, a loop iterates N times, but part of the loop is protected by a conditional statement and only executes M times, where N is an integer multiple of M. The code protected by the conditional is said to have an occurrence of N/M.

If N is pipelined with an initiation interval II, any function or loops protected by the conditional statement may be pipelined with a higher initiation interval than II (at a slower rate: this code is not executed as often) and can potentially be shared better within the enclosing higher rate pipeline.

Indentifying a region with an occurrence allows the functions and loops in this region to be pipelined with an initiation interval which is slower than the enclosing function or loop.

<*location*> - Specify the location which has a slower rate of execution.

Options

-cycle <*int*> - Specify the occurrence N/M, where N is the number of times the enclosing function or loop is executed and M is the number of times the conditional region is executed. N must be an integer multiple of M.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP occurrence cycle=<int>
```

Examples

In the following example, region `Cond_Region` in function `foo` has an occurrence of 4: it executes at a rate 4 times slower than the code which encompasses it.

```
set_directive_occurrence -cycle 4 foo/Cond_Region
#pragma AP occurrence cycle=4
```

set_directive_pipeline

Syntax

```
set_directive_pipeline [OPTIONS] <location>
```

Description

The `set_directive_pipeline` command specifies the details for pipelining:

- Function pipelining
- Loop pipelining

A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II). The default initiation interval is 1, which process a new input every clock cycle, or it can be specified by the **`-II`** option.

If High-Level Synthesis cannot create a design with the specified II it will issue a warning and create a design with the lowest possible II: this design can then be analyzed with the warning message to determine what steps must be taken to create a design which can satisfy the required initiation interval.

<*location*> - The name of the location, in the format function[/label], to be pipelined.

Options

`-II` <*integer*> - An integer specifying the desired initiation interval for the pipeline. High-Level Synthesis will try to meet this request: based on data dependencies, the actual result might have a larger II.

`-enable_flush` - This option implements a pipeline that can flush pipeline stages if the input of the pipeline stalls. This feature implements additional control logic, has greater area and is optional.

`-rewind` - This option is only applicable to a loops and enables rewinding, which enables continuous loop pipelining (with no pause between one loop iteration ending and the next starting). Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop is considered as initialization, will be executed only once in the pipeline and cannot contain any conditional operations (if-else).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP pipeline \
        II=<int> \
        enable_flush \
        rewind
```

Examples

In the following example, function `foo` is pipelined with an initiation interval of 1.

```
set_directive_pipeline foo
#pragma AP pipeline
```

Here, loop `loop_1` in function `foo` is pipelined with an initiation interval of 4 and pipelining flush is enabled.

```
set_directive_pipeline -II 4 -enable_flush foo/loop_1
#pragma AP pipeline II=4 enable_flush
```

set_directive_protocol

Syntax

set_directive_protocol [*OPTIONS*] <*location*>

Description

This commands specifies a region of the code, a protocol region, in which no clock operations will be inserted by High-Level Synthesis unless explicitly specified in the code. A protocol region can be used to manually specify an interface protocol: High-Level Synthesis will not insert any clocks between any operations including those which read from or write to function arguments. The order of read and writes will therefore be obeyed at the RTL.

A clock operation may be specified in C using an `ap_wait()` statement (include `ap_utils.h`) and may be specified in C++ and SystemC designs by using the `wait()` statement (include `systemc.h`). The `ap_wait` and `wait` statements have no effect on the simulation of C and C++ designs respectively: they are only interpreted by High-Level Synthesis.

A region of code can be created in the C code by enclosing the region in braces **{}** and naming it. (for example, `io_section:{..lines of C code...}` defines a region called `io_section`).

<*location*> - The name of the location, in the format function[/label], to be implemented in a cycle-accurate manner, corresponding to external protocol requirements.

Options

-mode (**floating** | **fixed**) - The default mode (**floating**) allows the code corresponding to statements outside the protocol region to overlap with the statements in the protocol statements in the final RTL: the protocol region remains cycle accurate but other operations can occur at the same time.

The fixed mode ensures there is no overlap.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP protocol \
        <floating, fixed>
```

Examples

This example defines region `io_section` in function `foo` as a fixed protocol region. The pragma should be placed inside of region `io_section`.

```
set_directive_protocol -mode fixed foo/io_section
#pragma AP protocol fixed
```

set_directive_resource

Syntax

set_directive_resource -core <*string*> <*location*> <*variable*>

Description

Specify that a specific library resource (core) be used to implement a variable (array, arithmetic operation or function argument) in the RTL.

High-Level Synthesis will implement the operations in the code using the cores available in the currently loaded library. When multiple cores in the library can be used to implement the variable, the `set_directive_resource` command specifies which core is used. Use the `list_core` command to list the available cores in the library. If no resource is specified, High-Level Synthesis will determine the resource to use.

The most common use of `set_directive_resource` is to specify which memory element in the library is used to implement an array. This allows you to control whether, for example, the array is implemented as a single or dual-port RAM. This usage is particularly important for arrays on the top-level function interface, since the memory associated with the array determines the ports in the RTL.

`<location>` - The location, in the format function[/label], where the variable can be found.

`<variable>` - Specify the name of the variable.

Options

`-core` `<string>` - Specify the name of the core, as defined in the technology library.

`-port_map` `<string>` - This option is used to specify port mappings when using the IP generation flow to map ports on the design with ports on the adapter. The argument to this option is a Tcl list of the design port and adapter ports.

`-metadata` `<string>` -This option is used to specify bus options when using the IP generation flow. The argument to this option is quoted list of bus operation directives.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP resource \
        variable=<variable> \
        core=<core>
```

Examples

In this example, variable `coeffs[128]` is an argument to top-level function `foo_top`. This directive specifies coeffs be implemented with core RAM_1P from the library. The ports created in the RTL to access the values of coeffs, will be those defined in the core RAM_1P.

```
set_directive_resource -core RAM_1P foo_top coeffs
#pragma AP resource variable=coeffs core=RAM_1P
```

Given code `Result=A*B` in function `foo`, this example specifies the multiplication be implemented with two-stage pipelined multiplier core, Mul2S.

```
set_directive_resource -core Mul2S foo Result
#pragma AP resource variable=Result core=Mul2S
```

set_directive_top

Syntax

set_directive_top [*OPTIONS*] <*location*>

Description

This directive attaches a name to a function, which can then be used for the `set_top` command. This is typically used to synthesize member functions of a class in C++.

The directive should be specified in an active solution and then the `set_top` command should be used with the new name.

<*location*> - The name of the function to be renamed.

Options

-name <*string*> - Specify the name to be used by the `set_top` command.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP top \
        name=<string>
```

Examples

In this example, function `foo_long_name` is renamed to `DESIGN_TOP`, which is then specified as the top-level. If the pragma is placed in the code, the `set_top` command must still be issued or the top-level specified in the GUI project settings.

```
set_directive_top -name DESIGN_TOP foo_long_name
#pragma AP top name=DESIGN_TOP
```

```
set_top DESIGN_TOP
```

set_directive_unroll

Syntax

set_directive_unroll [*OPTIONS*] <*location*>

Description

This command has been renamed `set_directive_loop_unroll` (same arguments and options as detailed here) and will be deprecated.

Transforms loop by creating multiples copies of the loop body.

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by any logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic, which represents the loop-body, which is executed for the same number of iterations.

The `set_directive_unroll` command allows the loop to be fully unrolled, creating as many copies of the loop-body in the RTL as there are loop iterations, or partially unrolled by a factor N, creating N copies of the loop body and adjusting the loop iteration accordingly.

If the factor N used for partial unrolling is not an integer multiple of the original loop iteration count, the original exit condition needs to be checked after each unrolled fragment of the loop body.

To unroll a loop completely, the loop bounds need to be known at compile time. This is not required for partial unrolling.

<*location*> - The location of the loop, in the format function[/label], to be unrolled.

Options

-factor <*integer*> - Non-zero integer indicating that partial unrolling is requested. The loop body will be repeated this number of times, and the iteration information will be adjusted accordingly.

-region - This option should be specified when seeking to unroll all loop within a loop without unrolling the enclosing loop itself.

Take the example where loop `loop_1` contains multiple loops at the same level of loop hierarchy, loops `loop_2` and `loop_3`. A named loop, such as `loop_1` is also a

region/location in the code: a section of code enclosed by braces **{ }**. If the unroll directive is specified on location `<function>/loop_1`, it will unroll `loop_1`.

The -region option specifies that the directive be applied only to the loops enclosed the named region: this results in `loop_1` being left rolled, but all loops inside it (`loop_2` and `loop_3`) being unrolled.

`-skip_exit_check` - This option is only effective if a factor is specified (partial unrolling).

- Fixed bounds - No exit condition check is performed if the iteration count is a multiple of the factor. For cases where the iteration count is not an integer multiple of the factor, unrolling will be prevented and a warning issued (the exit check must be performed in order to proceed).
- Variable bounds - The exit condition check is removed. The designer is responsible for ensuring the variable bounds is an integer multiple of the factor and that no exit check is in fact required.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP unroll \
        skip_exit_check \
        factor=<int> \
        region
```

Examples

This example unrolls loop `L1` in function `foo`. The pragma should be placed in the body of loop `L1`.

```
set_directive_unroll foo/L1
#pragma AP unroll
```

In this example, an unroll factor of 4 is specified on loop `L2` of function `foo` and the exit check is removed. The pragma should be placed in the body of loop `L2`.

```
set_directive_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

Here, all loops inside loop `L3` in function `foo` are unrolled, but not loop `L3` itself. The **`-region`** option specifies the location be considered an enclosing region and not a loop label.

```
set_directive_unroll -region foo/L3
```

```
#pragma AP unroll region
```

set_part

Syntax

set_part *<device_specification>*

Description

The `set_part` command sets a target device for the current solution. The command can only be executed in the context of an active solution.

< *device_specification*> - A device specification sets the target device for High-Level Synthesis synthesis and implementation.

<*device_family*> - The device specification can be simply the device family name, which will use the default device in the family.

<device><package><speed_grade> - The device specification can also be the target device name including device, package and speed-grade information.

Pragma

There is no pragma equivalent of the `set_part` command.

Examples

The FPGA libraries provided with High-Level Synthesis can be added to the current solution by simply providing the device family name as shown. In this case, the default device, package and speed-grade specified in the High-Level Synthesis FPGA library for this device family are used.

```
set_part virtex6
```

The FPGA libraries provided with High-Level Synthesis can optionally specify the specific device with package and speed-grade information.

```
set_part xc6vlx240tff1156-1
```

set_top

Syntax

```
set_top <top>
```

Description

The `set_top` command defines the top-level function to be synthesized. Any functions called from this function will also be part of the design.

<*top*> - Name of the function to be synthesized.

Pragma

There is no pragma equivalent of the `set_top` command.

Examples

This example sets the top-level function as `foo_top`.

```
set_top foo_top
```

XILINX

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

- Vivado Design Suite 2012.2 Documentation (http://www.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm)